Table of Contents

Introduction	1.1
Libraries	1.2
Station Class	1.3
Station Examples	1.4
Soft Access Point Class	1.5
Soft Access Point Examples	1.6
Scan Class	1.7
Scan Examples	1.8
Client Class	1.9
Client Examples	1.10
Client Secure Class	1.11
Client Secure Examples	1.12
Server Class	1.13
Server Examples	1.14
UDP Class	1.15
UDP Examples	1.16
Generic Class	1.17
Generic Examples	1.18

ESP8266 is all about Wi-Fi. If you are eager to connect your new ESP8266 module to Wi-Fi network to start sending and receiving data, this is a good place to start. If you are looking for more in depth details of how to program specific Wi-Fi networking functionality, you are also in the right place.

Table of Contents

- Introduction
  - Quick Start
  - Who is Who
- Class Description
  - Station
  - Soft Access Point
  - Scan
  - Client
  - Client Secure
  - Server
  - UDP
  - Generic
- Diagnostics
  - Check Return Codes
  - Use printDiag
  - Enable Wi-Fi Diagnostic
  - Enable Debugging in IDE
- What's Inside?

Introduction

The Wi-Fi library for ESP8266 has been developed basing on ESP8266 SDK, using naming convention and overall functionality philosophy of Arduino WiFi library. Over time the wealth Wi-Fi features ported from ESP9266 SDK to esp8266 / Adruino outgrow Arduino WiFi library and it became apparent that we need to provide separate documentation on what is new and extra.

This documentation will walk you through several classes, methods and properties of ESP8266WiFi library. If you are new to C++ and Arduino, don't worry. We will start from general concepts and then move to detailed description of members of each particular class including usage examples.

The scope of functionality offered by ESP8266WiFi library is quite extensive and therefore this description has been broken up into separate documents marked with :arrow_right:.

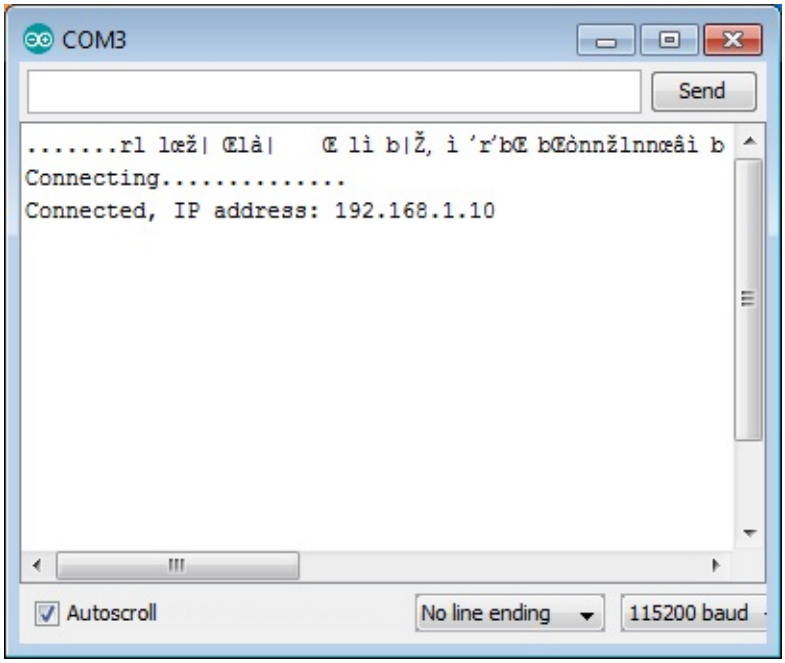
Quick Start

Hopefully you are already familiar how to load Blink.ino sketch to ESP8266 module and get the LED blinking. If not, please check this tutorial by Adafruit or another great tutorial developed by Sparkfun.

To hook up ESP module to Wi-Fi (like hooking up a mobile phone to a hot spot), you need just couple of lines of code:

```
#include <ESP8266WiFi.h>

void setup()
{
  Serial.begin(115200);
  Serial.println();

  WiFi.begin("network-name", "pass-to-network");

  Serial.print("Connecting");
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println();

  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {}
```

In the line `WiFi.begin("network-name", "pass-to-network")` replace `network-name` and `pass-to-network` with name and password to the Wi-Fi network you like to connect. Then upload this sketch to ESP module and open serial monitor. You should see something like:



How does it work? In the first line of sketch `#include <ESP8266WiFi.h>` we are including ESP8266WiFi library. This library provides ESP8266 specific Wi-Fi routines we are calling to connect to network.

Actual connection to Wi-Fi is initialized by calling:

```
WiFi.begin("network-name", "pass-to-network");
```

Connection process can take couple of seconds and we are checking for this to complete in the following loop:

```
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
```

The `while()` loop will keep looping while `WiFi.status()` is other than `WL_CONNECTED`. The loop will exit only if the status changes to `WL_CONNECTED`.

The last line will then print out IP address assigned to ESP module by DHCP:

```
Serial.println(WiFi.localIP());
```

If you don't see the last line but just more and more dots `.........`, then likely name or password to the Wi-Fi network in sketch is entered incorrectly. Verify name and password by connecting from scratch to this Wi-Fi a PC or a mobile phone.

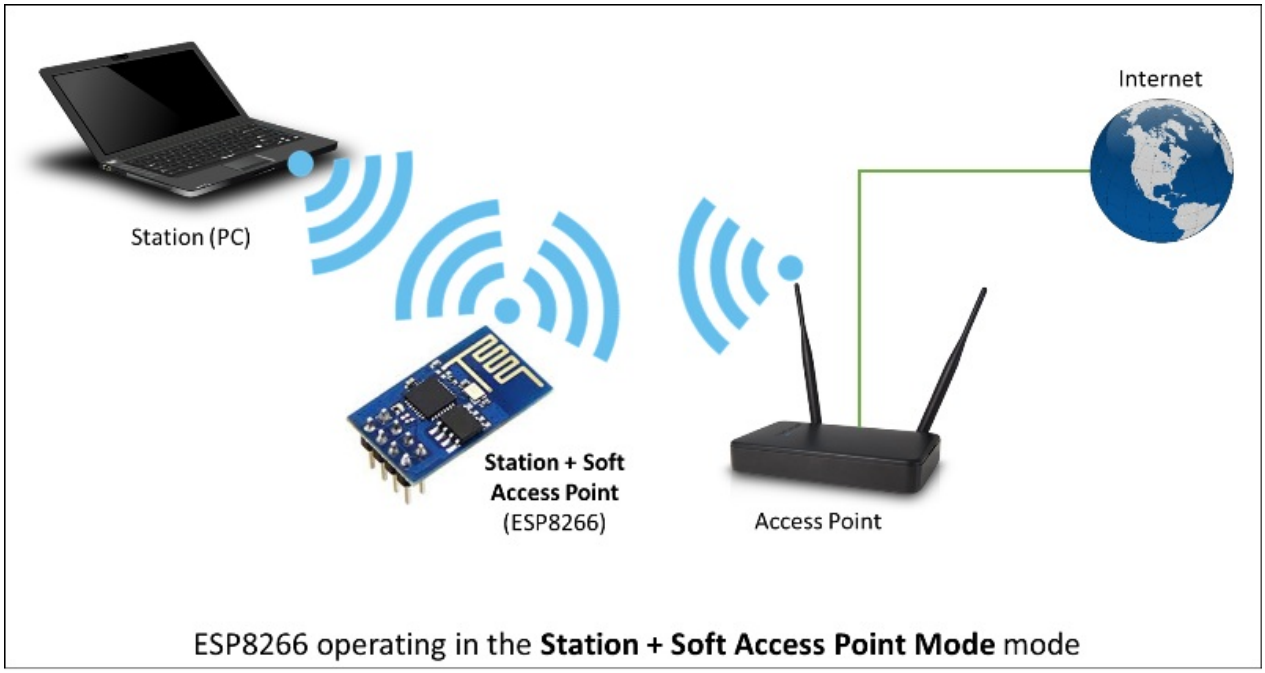
Note: if connection is established, and then lost for some reason, ESP will automatically reconnect to last used access point once it is again back on-line. This will be done automatically by Wi-Fi library, without any user intervention.

That's all you need to connect ESP8266 to Wi-Fi. In the following chapters we will explain what cool things can be done by ESP once connected.

Who is Who

Devices that connect to Wi-Fi network are called stations (STA). Connection to Wi-Fi is provided by an access point (AP), that acts as a hub for one or more stations. The access point on the other end is connected to a wired network. An access point is usually integrated with a router to provide access from Wi-Fi network to the internet. Each access point is recognized by a SSID (**S**ervice **S**et **ID**entifier), that essentially is the name of network you select when connecting a device (station) to the Wi-Fi.

ESP8266 module can operate as a station, so we can connect it to the Wi-Fi network. It can also operate as a soft access point (soft-AP), to establish its own Wi-Fi network. Therefore we can connect other stations to such ESP module. ESP8266 is also able to operate both in station and soft access point mode. This provides possibility of building e.g. mesh networks.



ESP8266 operating in the **Station + Soft Access Point Mode** mode

The ESP8266WiFi library provides wide collection of C++ methods) (functions) and properties) to configure and operate an ESP8266 module in station and / or soft access point mode. They are described in the following chapters.

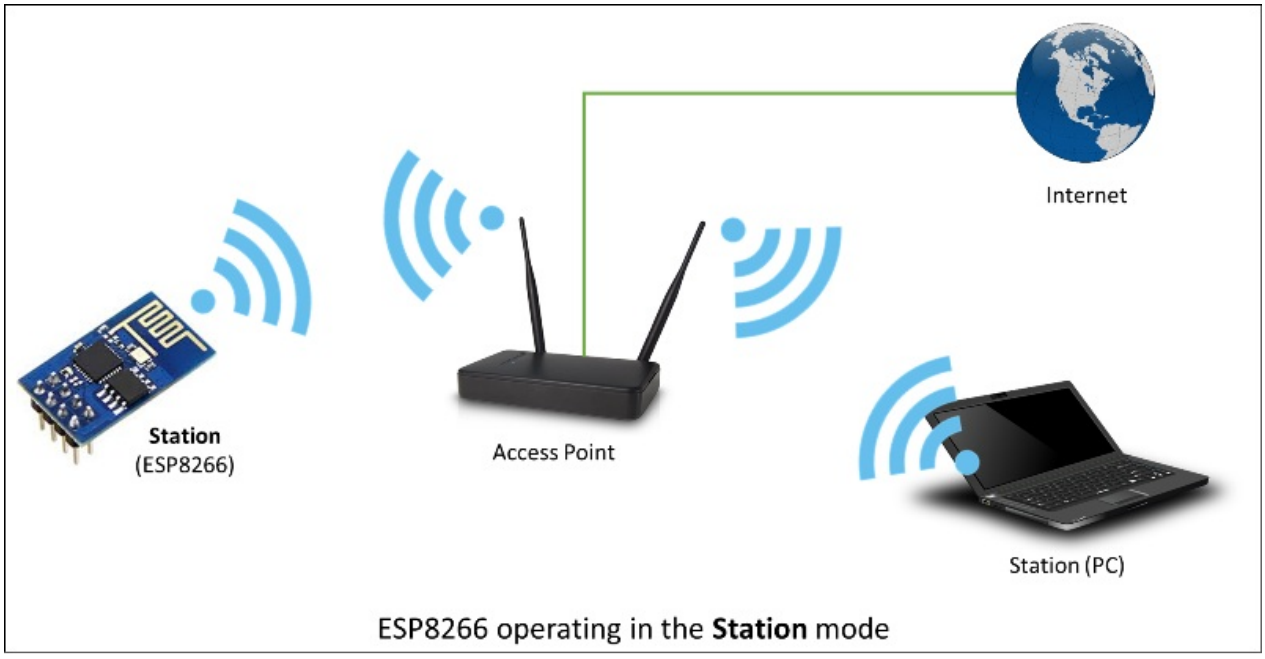
Class Description

The ESP8266WiFi library is broken up into several classes. In most of cases, when writing the code, user is not concerned with this classification. We are using it to break up description of this library into more manageable pieces.

Chapters below describe all function calls (methods) and properties) in C++ terms) listed in particular classes of ESP8266WiFi. Description is illustrated with application examples and code snippets to show how to use functions in practice. Most of this information is broken up into separate documents. Please follow :arrow_right: to access them.

Station

Station (STA) mode is used to get ESP module connected to a Wi-Fi network established by an access point.



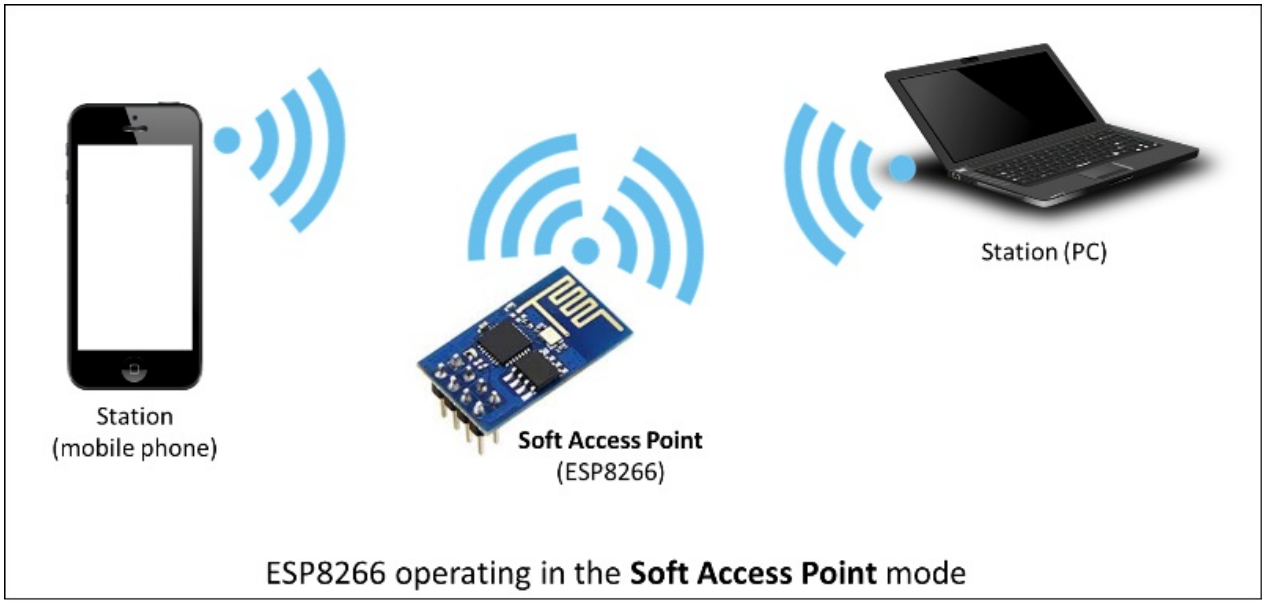
Station class has several features to facilitate management of Wi-Fi connection. In case the connection is lost, ESP8266 will automatically reconnect to the last used access point, once it is again available. The same happens on module reboot. This is possible since ESP is saving credentials to last used access point in flash (non-volatile) memory. Using the saved data ESP will also reconnect if sketch has been changed but code does not alter the Wi-Fi mode or credentials.

Station Class documentation :arrow_right: : begin | config | reconnect | disconnect | isConnected | setAutoConnect | getAutoConnect | setAutoReconnect | waitForConnectResult | macAddress | localIP | subnetMask | gatewayIP | dnsIP | hostname | status | SSID | psk | BSSID | RSSI | WPS | Smart Config

Check out separate section with examples :arrow_right:

Soft Access Point

An access point (AP) is a device that provides access to Wi-Fi network to other devices (stations) and connects them further to a wired network. ESP8266 can provide similar functionality except it does not have interface to a wired network. Such mode of operation is called soft access point (soft-AP). The maximum number of stations connected to the soft-AP is five.



ESP8266 operating in the **Soft Access Point** mode

The soft-AP mode is often used and an intermediate step before connecting ESP to a Wi-Fi in a station mode. This is when SSID and password to such network is not known upfront. ESP first boots in soft-AP mode, so we can connect to it using a laptop or a mobile phone. Then we are able to provide credentials to the target network. Once done ESP is switched to the station mode and can connect to the target Wi-Fi.

Another handy application of soft-AP mode is to set up mesh networks. ESP can operate in both soft-AP and Station mode so it can act as a node of a mesh network.

Soft Access Point Class documentation :arrow_right: : softAP | softAPConfig | softAPdisconnect | softAPgetStationNum | softAPIP | softAPmacAddress

Check out separate section with examples :arrow_right:

Scan

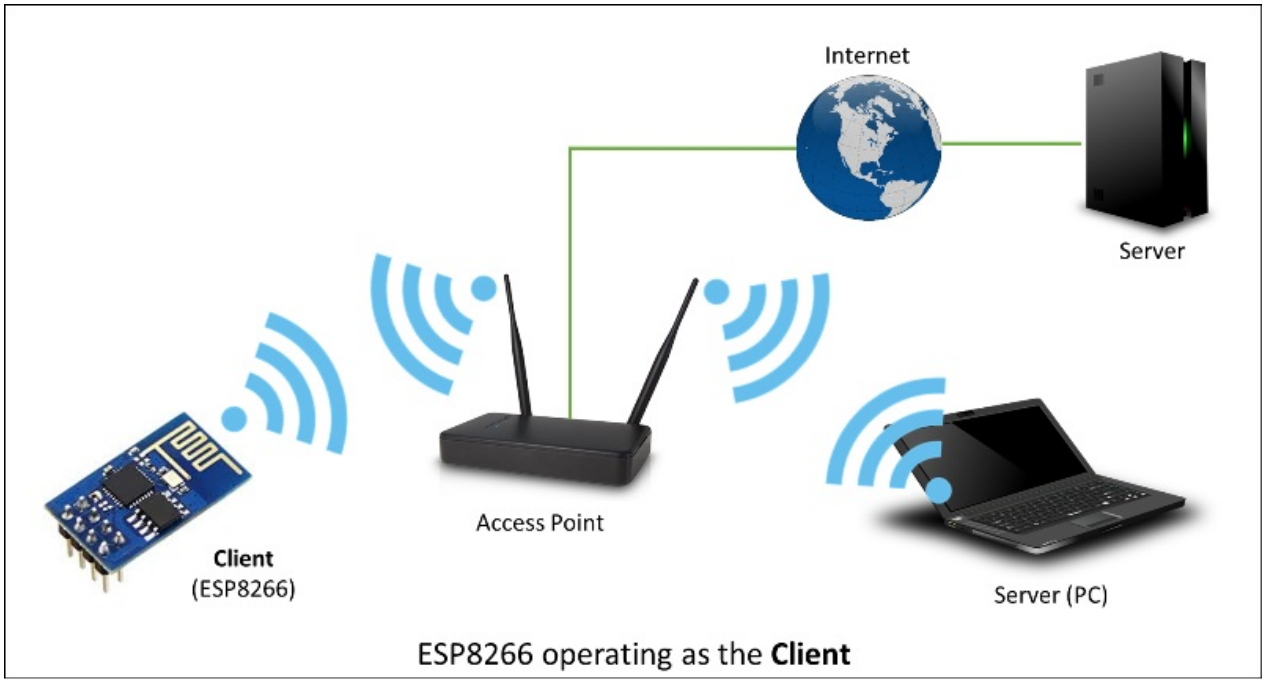
To connect a mobile phone to a hot spot, you typically open Wi-Fi settings app, list available networks and pick the hot spot you need. Then enter a password (or not) and you are in. You can do the same with ESP. Functionality of scanning for, and listing of available networks in range is implemented by the Scan Class.

Scan Class documentation :arrow_right: : scanNetworks | scanNetworksAsync | scanComplete | scanDelete | SSID | encryptionType | BSSID | BSSIDstr | channel | isHidden | getNetworkInfo

Check out separate section with examples :arrow_right:

Client

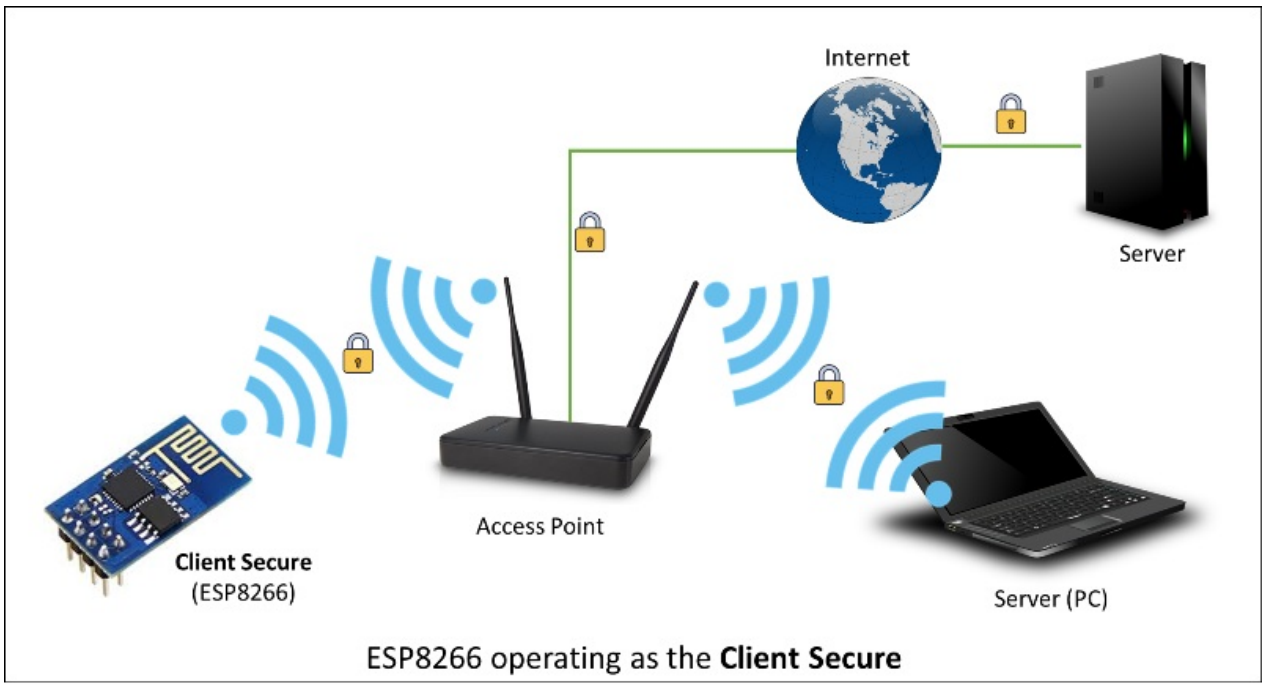
The Client class creates clients) that can access services provided by servers) in order to send, receive and process data.



Check out separate section with examples :arrow_right: / list of functions :arrow_right:

Client Secure

The Client Secure is an extension of Client Class where connection and data exchange with servers is done using a secure protocol. It supports TLS 1.1. The TLS 1.2 is not supported.



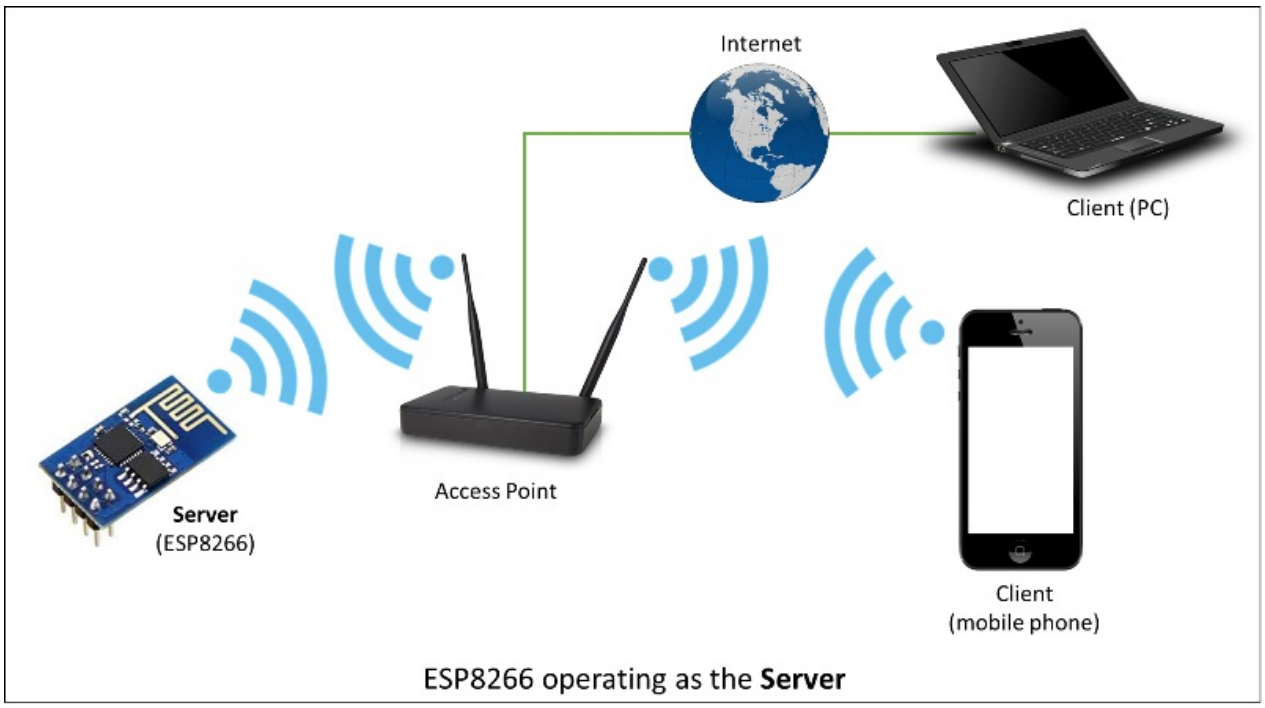
ESP8266 operating as the **Client Secure**

Secure applications have additional memory (and processing) overhead due to the need to run cryptography algorithms. The stronger the certificate's key, the more overhead is needed. In practice it is not possible to run more than a single secure client at a time. The problem concerns RAM memory we can not add, the flash memory size is usually not the issue. If you like to learn how client secure library has been developed, access to what servers have been tested, and how memory limitations have been overcame, read fascinating issue report #43.

Check out separate section with examples :arrow_right: / list of functions :arrow_right:

Server

The Server Class creates servers) that provide functionality to other programs or devices, called clients).



Clients connect to sever to send and receive data and access provided functionality.

Check out separate section with examples :arrow_right: / list of functions :arrow_right:

UDP

The UDP Class enables the User Datagram Protocol (UDP) messages to be sent and received. The UDP uses a simple "fire and forget" transmission model with no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

Check out separate section with examples :arrow_right: / list of functions :arrow_right:

Generic

There are several functions offered by ESP8266's SDK and not present in Arduino WiFi library. If such function does not fit into one of classes discussed above, it will likely be in Generic Class. Among them is handler to manage Wi-Fi events like connection, disconnection or obtaining an IP, Wi-Fi mode changes, functions to manage module sleep mode, hostname to an IP address resolution, etc.

Check out separate section with examples :arrow_right: / list of functions :arrow_right:

Diagnostics

There are several techniques available to diagnose and troubleshoot issues with getting connected to Wi-Fi and keeping connection alive.

Check Return Codes

Almost each function described in chapters above returns some diagnostic information.

Such diagnostic may be provided as a simple `boolean` type `true' or` false` to indicate operation result. You may check this result as described in examples, for instance:

```
Serial.printf("Wi-Fi mode set to WIFI_STA %s\n", WiFi.mode(WIFI_STA) ? "" : "Failed!")
;
```

Some functions provide more than just a binary status information. A good example is `WiFi.status()` .

```
Serial.printf("Connection status: %d\n", WiFi.status());
```

This function returns following codes to describe what is going on with Wi-Fi connection:

- 0 : `WL_IDLE_STATUS` when Wi-Fi is in process of changing between statuses
- 1 : `WL_NO_SSID_AVAIL` in case configured SSID cannot be reached
- 3 : `WL_CONNECTED` after successful connection is established
- 4 : `WL_CONNECT_FAILED` if password is incorrect
- 6 : `WL_DISCONNECTED` if module is not configured in station mode

It is a good practice to display and check information returned by functions. Application development and troubleshooting will be easier with that.

Use printDiag

There is a specific function available to print out key Wi-Fi diagnostic information:

```
WiFi.printDiag(Serial);
```

A sample output of this function looks as follows:

```
Mode: STA+AP
PHY mode: N
Channel: 11
AP id: 0
Status: 5
Auto connect: 1
SSID (10): sensor-net
Passphrase (12): 123!$#0&*esP
BSSID set: 0
```

Use this function to provide snapshot of Wi-Fi status in these parts of application code, that you suspect may be failing.

Enable Wi-Fi Diagnostic

By default the diagnostic output from Wi-Fi libraries is disabled when you call `Serial.begin`. To enable debug output again, call `Serial.setDebugOutput(true)`. To redirect debug output to `Serial1` instead, call `Serial1.setDebugOutput(true)`. For additional details regarding diagnostics using serial ports please refer to documentation.

Below is an example of output for sample sketch discussed in Quick Start above with `Serial.setDebugOutput(true)`:

```
Connectingscandone
state: 0 -> 2 (b0)
state: 2 -> 3 (0)
state: 3 -> 5 (10)
add 0
aid 1
cnt

connected with sensor-net, channel 6
dhcp client start...
chg_B1:-40
...ip:192.168.1.10,mask:255.255.255.0,gw:192.168.1.9
.
Connected, IP address: 192.168.1.10
```

The same sketch without `Serial.setDebugOutput(true)` will print out only the following:

```
Connecting....
Connected, IP address: 192.168.1.10
```

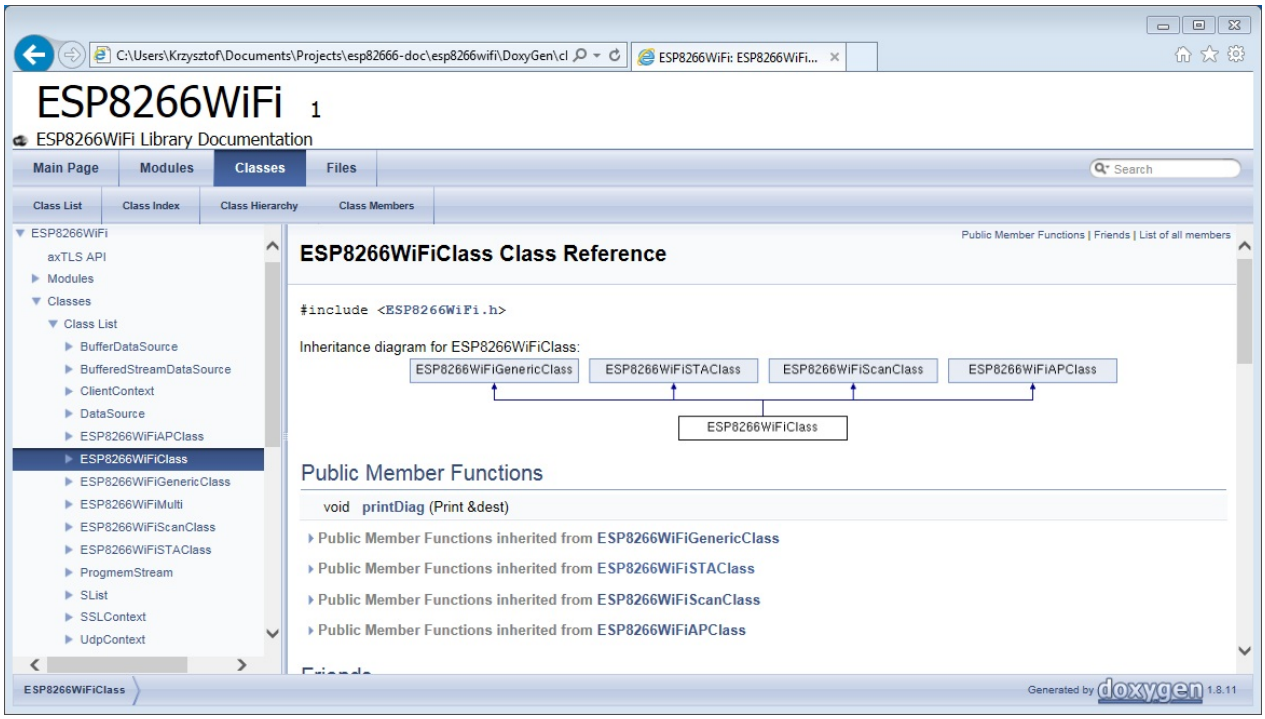
Enable Debugging in IDE

Arduino IDE provides convenient method to enable debugging for specific libraries.

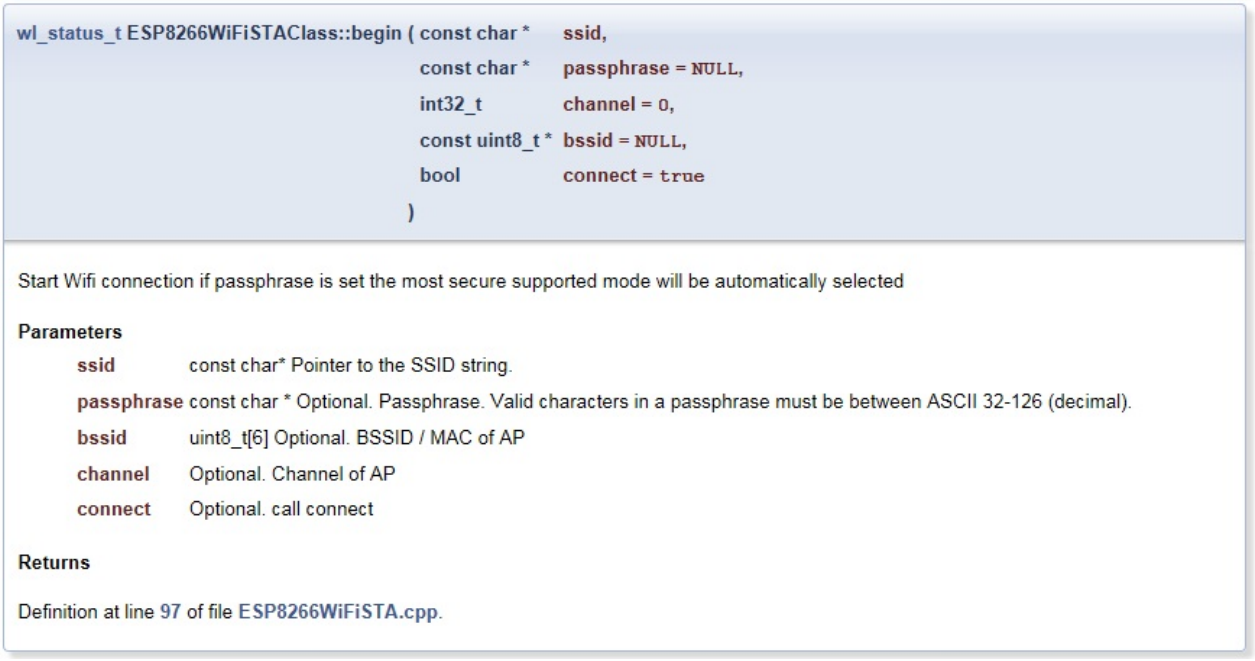
What's Inside?

If you like to analyze in detail what is inside of the ESP8266WiFi library, go directly to the ESP8266WiFi folder of esp8266 / Arduino repository on the GitHub.

To make the analysis easier, rather than looking into individual header or source files, use one of free tools to automatically generate documentation. The class index in chapter Class Description above has been prepared in no time using great Doxygen, that is the de facto standard tool for generating documentation from annotated C++ sources.



The tool crawls through all header and source files collecting information from formatted comment blocks. If developer of particular class annotated the code, you will see it like in examples below.



wl_status_t ESP8266WiFiSTAClass::begin (const char * ssid,
const char * passphrase = NULL,
int32_t channel = 0,
const uint8_t * bssid = NULL,
bool connect = true
)

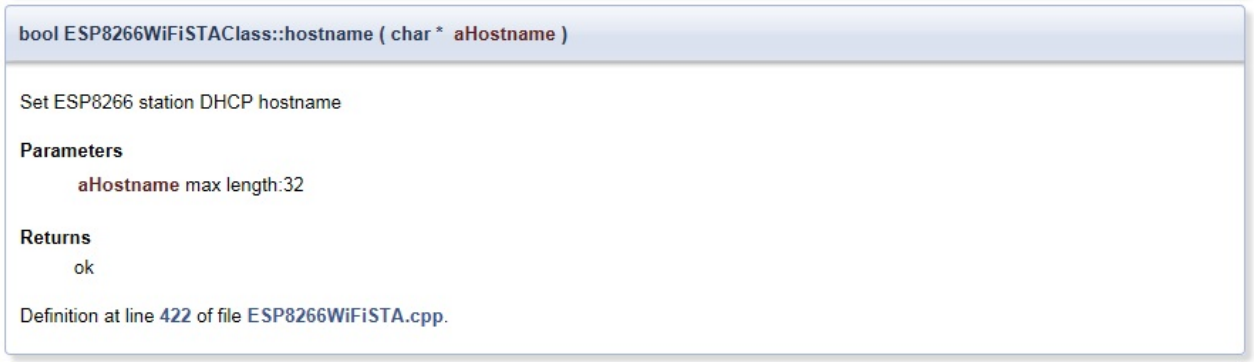
Start Wifi connection if passphrase is set the most secure supported mode will be automatically selected

Parameters

ssid	const char* Pointer to the SSID string.
passphrase	const char * Optional. Passphrase. Valid characters in a passphrase must be between ASCII 32-126 (decimal).
bssid	uint8_t[6] Optional. BSSID / MAC of AP
channel	Optional. Channel of AP
connect	Optional. call connect

Returns

Definition at line 97 of file ESP8266WiFiSTA.cpp.



bool ESP8266WiFiSTAClass::hostname (char * aHostname)

Set ESP8266 station DHCP hostname

Parameters

aHostname max length:32

Returns

ok

Definition at line 422 of file ESP8266WiFiSTA.cpp.

If code is not annotated, you will still see the function prototype including types of arguments, and can use provided links to jump straight to the source code to check it out on your own. Doxygen provides really excellent navigation between members of library.

uint8_t WiFiUDP::begin (uint16_t port) virtual

Definition at line 77 of file WiFiUdp.cpp.

Several classes of ESP8266WiFi are not annotated. When preparing this document, Doxygen has been tremendous help to quickly navigate through almost 30 files that make this library.

Table of Contents

- WiFi(ESP8266WiFi library)
- Ticker
- EEPROM
- I2C (Wire library)
- SPI
- SoftwareSerial
- ESP-specific APIs
- mDNS and DNS-SD responder (ESP8266mDNS library)
- SSDP responder (ESP8266SSDP)
- DNS server (DNSServer library)
- Servo
- Other libraries (not included with the IDE)

WiFi(ESP8266WiFi library)

The Wi-Fi library for ESP8266 has been developed basing on ESP8266 SDK, using naming convention and overall functionality philosophy of Arduino WiFi library. Over time the wealth Wi-Fi features ported from ESP9266 SDK to esp8266 / Adruino outgrow Arduino WiFi library and it became apparent that we need to provide separate documentation on what is new and extra.

ESP8266WiFi library documentation : Quick Start | Who is Who | Station | Soft Access Point | Scan | Client | Client Secure | Server | UDP | Generic | Diagnostics

Ticker

Library for calling functions repeatedly with a certain period. Two examples included.

It is currently not recommended to do blocking IO operations (network, serial, file) from Ticker callback functions. Instead, set a flag inside the ticker callback and check for that flag inside the loop function.

Here is library to simplificate `Ticker` usage and avoid WDT reset: TickerScheduler

EEPROM

This is a bit different from standard EEPROM class. You need to call `EEPROM.begin(size)` before you start reading or writing, size being the number of bytes you want to use. Size can be anywhere between 4 and 4096 bytes.

`EEPROM.write` does not write to flash immediately, instead you must call `EEPROM.commit()` whenever you wish to save changes to flash. `EEPROM.end()` will also commit, and will release the RAM copy of EEPROM contents.

EEPROM library uses one sector of flash located just after the SPIFFS.

Three examples included.

I2C (Wire library)

Wire library currently supports master mode up to approximately 450KHz. Before using I2C, pins for SDA and SCL need to be set by calling `Wire.begin(int sda, int scl)`, i.e. `Wire.begin(0, 2)` on ESP-01, else they default to pins 4(SDA) and 5(SCL).

SPI

SPI library supports the entire Arduino SPI API including transactions, including setting phase (CPHA). Setting the Clock polarity (CPOL) is not supported, yet (SPI_MODE2 and SPI_MODE3 not working).

SoftwareSerial

An ESP8266 port of SoftwareSerial library done by Peter Lerup (@plerup) supports baud rate up to 115200 and multiples SoftwareSerial instances. See https://github.com/plerup/espsoftwareserial if you want to suggest an improvement or open an issue related to SoftwareSerial.

ESP-specific APIs

APIs related to deep sleep and watchdog timer are available in the `ESP` object, only available in Alpha version.

`ESP.deepSleep(microseconds, mode)` will put the chip into deep sleep. `mode` is one of `WAKE_RF_DEFAULT`, `WAKE_RFCAL`, `WAKE_NO_RFCAL`, `WAKE_RF_DISABLED`. (GPIO16 needs to be tied to RST to wake from deepSleep.)

`ESP.rtcUserMemoryWrite(offset, &data, sizeof(data))` and `ESP.rtcUserMemoryRead(offset, &data, sizeof(data))` allow data to be stored in and retrieved from the RTC user memory of the chip respectively. Total size of RTC user memory is 512 bytes, so offset + sizeof(data) shouldn't exceed 512. Data should be 4-byte aligned. The stored data can be retained between deep sleep cycles. However, the data might be lost after power cycling the chip.

`ESP.restart()` restarts the CPU.

`ESP.getResetReason()` returns String containing the last reset resaon in human readable format.

`ESP.getFreeHeap()` returns the free heap size.

`ESP.getChipId()` returns the ESP8266 chip ID as a 32-bit integer.

Several APIs may be used to get flash chip info:

`ESP.getFlashChipId()` returns the flash chip ID as a 32-bit integer.

`ESP.getFlashChipSize()` returns the flash chip size, in bytes, as seen by the SDK (may be less than actual size).

`ESP.getFlashChipRealSize()` returns the real chip size, in bytes, based on the flash chip ID.

`ESP.getFlashChipSpeed(void)` returns the flash chip frequency, in Hz.

`ESP.getCycleCount()` returns the cpu instruction cycle count since start as an unsigned 32-bit. This is useful for accurate timing of very short actions like bit banging.

`ESP.getVcc()` may be used to measure supply voltage. ESP needs to reconfigure the ADC at startup in order for this feature to be available. Add the following line to the top of your sketch to use `getVcc`:

```
ADC_MODE(ADC_VCC);
```

TOUT pin has to be disconnected in this mode.

Note that by default ADC is configured to read from TOUT pin using `analogRead(A0)`, and `ESP.getVCC()` is not available.

mDNS and DNS-SD responder (ESP8266mDNS library)

Allows the sketch to respond to multicast DNS queries for domain names like "foo.local", and DNS-SD (service dicovery) queries. See attached example for details.

SSDP responder (ESP8266SSDP)

SSDP is another service discovery protocol, supported on Windows out of the box. See attached example for reference.

DNS server (DNSServer library)

Implements a simple DNS server that can be used in both STA and AP modes. The DNS server currently supports only one domain (for all other domains it will reply with NXDOMAIN or custom status code). With it clients can open a web server running on ESP8266 using a domain name, not an IP address. See attached example for details.

Servo

This library exposes the ability to control RC (hobby) servo motors. It will support upto 24 servos on any available output pin. By defualt the first 12 servos will use Timer0 and currently this will not interfere with any other support. Servo counts above 12 will use Timer1 and features that use it will be effected. While many RC servo motors will accept the 3.3V IO data pin from a ESP8266, most will not be able to run off 3.3v and will require another power source that matches their specifications. Make sure to connect the grounds between the ESP8266 and the servo motor power supply.

Other libraries (not included with the IDE)

Libraries that don't rely on low-level access to AVR registers should work well. Here are a few libraries that were verified to work:

- Adafruit_ILI9341 - Port of the Adafruit ILI9341 for the ESP8266
- arduinoVNC - VNC Client for Arduino
- arduinoWebSockets - WebSocket Server and Client compatible with ESP8266 (RFC6455)
- aREST - REST API handler library.
- Blynk - easy IoT framework for Makers (check out the Kickstarter page).
- DallasTemperature
- DHT-sensor-library - Arduino library for the DHT11/DHT22 temperature and humidity sensors. Download latest v1.1.1 library and no changes are necessary. Older versions should initialize DHT as follows: `DHT dht(DHTPIN, DHTTYPE, 15)`
- DimSwitch - Control electronic dimmable ballasts for fluorescent light tubes remotely as

if using a wall switch.

- Encoder - Arduino library for rotary encoders. Version 1.4 supports ESP8266.
- esp8266_mdns - mDNS queries and responses on esp8266. Or to describe it another way: An mDNS Client or Bonjour Client library for the esp8266.
- ESPAsyncTCP - Asynchronous TCP Library for ESP8266 and ESP32/31B
- ESPAsyncWebServer - Asynchronous Web Server Library for ESP8266 and ESP32/31B
- Homie for ESP8266 - Arduino framework for ESP8266 implementing Homie, an MQTT convention for the IoT.
- NeoPixel - Adafruit's NeoPixel library, now with support for the ESP8266 (use version 1.0.2 or higher from Arduino's library manager).
- NeoPixelBus - Arduino NeoPixel library compatible with ESP8266. Use the "DmaDriven" or "UartDriven" branches for ESP8266. Includes HSL color support and more.
- PubSubClient - MQTT library by @Imroy.
- RTC - Arduino Library for Ds1307 & Ds3231 compatible with ESP8266.
- Souliss, Smart Home - Framework for Smart Home based on Arduino, Android and openHAB.
- ST7735 - Adafruit's ST7735 library modified to be compatible with ESP8266. Just make sure to modify the pins in the examples as they are still AVR specific.
- Task - Arduino Nonpreemptive multitasking library. While similiar to the included Ticker library in the functionality provided, this library was meant for cross Arduino compatibility.
- TickerScheduler - Library provides simple scheduler for `Ticker` to avoid WDT reset
- Teleinfo - Generic French Power Meter library to read Teleinfo energy monitoring data such as consuption, contract, power, period, ... This library is cross platform, ESP8266, Arduino, Particle, and simple C++. French dedicated post on author's blog and all related information about Teleinfo also available.
- UTFT-ESP8266 - UTFT display library with support for ESP8266. Only serial interface (SPI) displays are supported for now (no 8-bit parallel mode, etc). Also includes support for the hardware SPI controller of the ESP8266.
- WiFiManager - WiFi Connection manager with web captive portal. If it can't connect, it starts AP mode and a configuration portal so you can choose and enter WiFi credentials.
- OneWire - Library for Dallas/Maxim 1-Wire Chips.
- Adafruit-PCD8544-Nokia-5110-LCD-Library - Port of the Adafruit PCD8544 - library for the ESP8266.
- PCF8574_ESP - A very simplistic library for using the PCF8574/PCF8574A I2C 8-pin GPIO-expander.
- Dot Matrix Display Library 2 - Freetronics DMD & Generic 16 x 32 P10 style Dot Matrix

Display Library

- SdFat-beta - SD-card library with support for long filenames, software- and hardware-based SPI and lots more.
- FastLED - a library for easily & efficiently controlling a wide variety of LED chipsets, like the Neopixel (WS2812B), DotStar, LPD8806 and many more. Includes fading, gradient, color conversion functions.
- OLED - a library for controlling I2C connected OLED displays. Tested with 0.96 inch OLED graphics display.
- MFRC522 - A library for using the Mifare RC522 RFID-tag reader/writer.
- Ping - lets the ESP8266 ping a remote machine.

ESP8266WiFi Library :back:

Station Class

The number of features provided by ESP8266 in the station mode is far more extensive than covered in original Arduino WiFi library. Therefore, instead of supplementing original documentation, we have decided to write a new one from scratch.

Description of station class has been broken down into four parts. First discusses methods to establish connection to an access point. Second provides methods to manage connection like e.g. `reconnect` or `isConnected`. Third covers properties to obtain information about connection like MAC or IP address. Finally the fourth section provides alternate methods to connect like e.g. Wi-Fi Protected Setup (WPS).

Table of Contents

- Start Here
  - begin
  - config
- Manage Connection
  - reconnect
  - disconnect
  - isConnected
  - setAutoConnect
  - getAutoConnect
  - setAutoReconnect
  - waitForConnectResult
- Configuration
  - macAddress
  - localIP
  - subnetMask
  - gatewayIP
  - dnsIP
  - hostname
  - status
  - SSID
  - psk
  - BSSID
  - RSSI

- Connect Different
  - WPS
  - Smart Config

Points below provide description and code snippets how to use particular methods.

For more code samples please refer to separate section with examples :arrow_right: dedicated specifically to the Station Class.

Start Here

Switching the module to Station mode is done with `begin` function. Typical parameters passed to `begin` include SSID and password, so module can connect to specific Access Point.

```
WiFi.begin(ssid, password)
```

By default, ESP will attempt to reconnect to Wi-Fi network whenever it is disconnected. There is no need to handle this by separate code. A good way to simulate disconnection would be to reset the access point. ESP will report disconnection, and then try to reconnect automatically.

begin

There are several version (called *function overloads* in C++) of `begin` function. One was presented just above: `WiFi.begin(ssid, password)`. Overloads provide flexibility in number or type of accepted parameters.

The simplest overload of `begin` is as follows:

```
WiFi.begin()
```

Calling it will instruct module to switch to the station mode and connect to the last used access point basing on configuration saved in flash memory.

Below is the syntax of another overload of `begin` with the all possible parameters:

```
WiFi.begin(ssid, password, channel, bssid, connect)
```

Meaning of parameters is as follows:

- `ssid` - a character string containing the SSID of Access Point we would like to connect

to, may have up to 32 characters
- `password` to the access point, a character string that should be minimum 8 characters long and not longer than 64 characters
- `channel` of AP, if we like to operate using specific channel, otherwise this parameter may be omitted
- `bssid` - mac address of AP, this parameter is also optional
- `connect` - a `boolean` parameter that if set to `false`, will instruct module just to save the other parameters without actually establishing connection to the access point

config

Disable DHCP client (Dynamic Host Configuration Protocol) and set the IP configuration of station interface to user defined arbitrary values. The interface will be a static IP configuration instead of values provided by DHCP.

```
WiFi.config(local_ip, gateway, subnet, dns1, dns2)
```

Function will return `true` if configuration change is applied successfully. If configuration can not be applied, because e.g. module is not in station or station + soft access point mode, then `false` will be returned.

The following IP configuration may be provided:

- `local_ip` - enter here IP address you would like to assign the ESP station's interface
- `gateway` - should contain IP address of gateway (a router) to access external networks
- `subnet` - this is a mask that defines the range of IP addresses of the local network
- `dns1`, `dns2` - optional parameters that define IP addresses of Domain Name Servers (DNS) that maintain a directory of domain names (like e.g. *www.google.co.uk*) and translate them for us to IP addresses

Example code:

```
#include <ESP8266WiFi.h>

const char* ssid = "********";
const char* password = "********";

IPAddress staticIP(192,168,1,22);
IPAddress gateway(192,168,1,9);
IPAddress subnet(255,255,255,0);

void setup(void)
{
  Serial.begin(115200);
  Serial.println();

  Serial.printf("Connecting to %s\n", ssid);
  WiFi.begin(ssid, password);
  WiFi.config(staticIP, gateway, subnet);
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {}
```

Example output:

```
Connecting to sensor-net
.
Connected, IP address: 192.168.1.22
```

Please note that station with static IP configuration usually connects to the network faster. In the above example it took about 500ms (one dot `.` displayed). This is because obtaining of IP configuration by DHCP client takes time and in this case this step is skipped.

Manage Connection

reconnect

Reconnect the station. This is done by disconnecting from the access point an then initiating connection back to the same AP.

```
WiFi.reconnect()
```

Notes:

1. Station should be already connected to an access point. If this is not the case, then function will return `false` not performing any action.
2. If `true` is returned it means that connection sequence has been successfully started. User should still check for connection status, waiting until `WL_CONNECTED` is reported:

```
WiFi.reconnect();
while (WiFi.status() != WL_CONNECTED)
{
  delay(500);
  Serial.print(".");
}
```

disconnect

Sets currently configured SSID and password to `null` values and disconnects the station from an access point.

```
WiFi.disconnect(wifioff)
```

The `wifioff` is an optional `boolean` parameter. If set to `true`, then the station mode will be turned off.

isConnected

Returns `true` if Station is connected to an access point or `false` if not.

```
WiFi.isConnected()
```

setAutoConnect

Configure module to automatically connect on power on to the last used access point.

```
WiFi.setAutoConnect(autoConnect)
```

The `autoConnect` is an optional parameter. If set to `false` then auto connection functionality up will be disabled. If omitted or set to `true`, then auto connection will be enabled.

getAutoConnect

This is "companion" function to `setAutoConnect()`. It returns `true` if module is configured to automatically connect to last used access point on power on.

```
WiFi.getAutoConnect()
```

If auto connection functionality is disabled, then function returns `false`.

setAutoReconnect

Set whether module will attempt to reconnect to an access point in case it is disconnected.

```
WiFi.setAutoReconnect(autoReconnect)
```

If parameter `autoReconnect` is set to `true`, then module will try to reestablish lost connection to the AP. If set to `false` then module will stay disconnected.

Note: running `setAutoReconnect(true)` when module is already disconnected will not make it reconnect to the access point. Instead `reconnect()` should be used.

waitForConnectResult

Wait until module connects to the access point. This function is intended for module configured in station or station + soft access point mode.

```
WiFi.waitForConnectResult()
```

Function returns one of the following connection statuses:

- `WL_CONNECTED` after successful connection is established
- `WL_NO_SSID_AVAIL` in case configured SSID cannot be reached
- `WL_CONNECT_FAILED` if password is incorrect
- `WL_IDLE_STATUS` when Wi-Fi is in process of changing between statuses
- `WL_DISCONNECTED` if module is not configured in station mode

Configuration

macAddress

Get the MAC address of the ESP station's interface.

```
WiFi.macAddress(mac)
```

Function should be provided with `mac` that is a pointer to memory location (an `uint8_t` array the size of 6 elements) to save the mac address. The same pointer value is returned by the function itself.

Example code:

```
if (WiFi.status() == WL_CONNECTED)
{
  uint8_t macAddr[6];
  WiFi.macAddress(macAddr);
  Serial.printf("Connected, mac address: %02x:%02x:%02x:%02x:%02x:%02x\n", macAddr[0],
 macAddr[1], macAddr[2], macAddr[3], macAddr[4], macAddr[5]);
}
```

Example output:

```
Mac address: 5C:CF:7F:08:11:17
```

If you do not feel comfortable with pointers, then there is optional version of this function available. Instead of the pointer, it returns a formatted `String` that contains the same mac address.

```
WiFi.macAddress()
```

Example code:

```
if (WiFi.status() == WL_CONNECTED)
{
  Serial.printf("Connected, mac address: %s\n", WiFi.macAddress().c_str());
}
```

localIP

Function used to obtain IP address of ESP station's interface.

```
WiFi.localIP()
```

The type of returned value is IPAddress. There is a couple of methods available to display this type of data. They are presented in examples below that cover description of `subnetMask`, `gatewayIP` and `dnsIP` that return the IPAdress as well.

Example code:

```
if (WiFi.status() == WL_CONNECTED)
{
  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}
```

Example output:

```
Connected, IP address: 192.168.1.10
```

subnetMask

Get the subnet mask of the station's interface.

```
WiFi.subnetMask()
```

Module should be connected to the access point to obtain the subnet mask.

Example code:

```
Serial.print("Subnet mask: ");
Serial.println(WiFi.subnetMask());
```

Example output:

```
Subnet mask: 255.255.255.0
```

gatewayIP

Get the IP address of the gateway.

```
WiFi.gatewayIP()
```

Example code:

```
Serial.printf("Gataway IP: %s\n", WiFi.gatewayIP().toString().c_str());
```

Example output:

```
Gataway IP: 192.168.1.9
```

dnsIP

Get the IP addresses of Domain Name Servers (DNS).

```
WiFi.dnsIP(dns_no)
```

With the input parameter `dns_no` we can specify which Domain Name Server's IP we need. This parameter is zero based and allowed values are none, 0 or 1. If no parameter is provided, then IP of DNS #1 is returned.

Example code:

```
Serial.print("DNS #1, #2 IP: ");
WiFi.dnsIP().printTo(Serial);
Serial.print(", ");
WiFi.dnsIP(1).printTo(Serial);
Serial.println();
```

Example output:

```
DNS #1, #2 IP: 62.179.1.60, 62.179.1.61
```

hostname

Get the DHCP hostname assigned to ESP station.

```
WiFi.hostname()
```

Function returns `String` type. Default hostname is in format `ESP_24xMAC` where 24xMAC are the last 24 bits of module's MAC address.

The hostname may be changed using the following function:

```
WiFi.hostname(aHostname)
```

Input parameter `aHostname` may be a type of `char*`, `const char*` or `String`. Maximum length of assigned hostname is 32 characters. Function returns either `true` or `false` depending on result. For instance, if the limit of 32 characters is exceeded, function will return `false` without assigning the new hostname.

Example code:

```
Serial.printf("Default hostname: %s\n", WiFi.hostname().c_str());
WiFi.hostname("Station_Tester_02");
Serial.printf("New hostname: %s\n", WiFi.hostname().c_str());
```

Example output:

```
Default hostname: ESP_081117
New hostname: Station_Tester_02
```

status

Return the status of Wi-Fi connection.

```
WiFi.status()
```

Function returns one of the following connection statuses:

- `WL_CONNECTED` after successful connection is established
- `WL_NO_SSID_AVAIL` in case configured SSID cannot be reached
- `WL_CONNECT_FAILED` if password is incorrect
- `WL_IDLE_STATUS` when Wi-Fi is in process of changing between statuses
- `WL_DISCONNECTED` if module is not configured in station mode

Returned value is type of `wl_status_t` defined in wl_definitions.h

Example code:

```
#include <ESP8266WiFi.h>

void setup(void)
{
  Serial.begin(115200);
  Serial.printf("Connection status: %d\n", WiFi.status());
  Serial.printf("Connecting to %s\n", ssid);
  WiFi.begin(ssid, password);
  Serial.printf("Connection status: %d\n", WiFi.status());
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.printf("\nConnection status: %d\n", WiFi.status());
  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {}
```

Example output:

```
Connection status: 6
Connecting to sensor-net
Connection status: 6
......
Connection status: 3
Connected, IP address: 192.168.1.10
```

Particular connection statuses 6 and 3 may be looked up in wl_definitions.h as follows:

```
3 - WL_CONNECTED
6 - WL_DISCONNECTED
```

Basing on this example, when running above code, module is initially disconnected from the network and returns connection status `6 - WL_DISCONNECTED`. It is also disconnected immediately after running `WiFi.begin(ssid, password)`. Then after about 3 seconds (basing on number of dots displayed every 500ms), it finally gets connected returning status `3 - WL_CONNECTED`.

SSID

Return the name of Wi-Fi network, formally called Service Set Identification (SSID).

```
WiFi.SSID()
```

Returned value is of the `String` type.

Example code:

```
Serial.printf("SSID: %s\n", WiFi.SSID().c_str());
```

Example output:

```
SSID: sensor-net
```

psk

Return current pre shared key (password) associated with the Wi-Fi network.

```
WiFi.psk()
```

Function returns value of the `String` type.

BSSID

Return the mac address the access point where ESP module is connected to. This address is formally called Basic Service Set Identification (BSSID).

```
WiFi.BSSID()
```

The `BSSID()` function returns a pointer to the memory location (an `uint8_t` array with the size of 6 elements) where the BSSID is saved.

Below is similar function, but returning BSSID but as a `String` type.

```
WiFi.BSSIDstr()
```

Example code:

```
Serial.printf("BSSID: %s\n", WiFi.BSSIDstr().c_str());
```

Example output:

```
BSSID: 00:1A:70:DE:C1:68
```

RSSI

Return the signal strength of Wi-Fi network, that is formally called Received Signal Strength Indication (RSSI).

```
WiFi.RSSI()
```

Signal strength value is provided in dBm. The type of returned value is `int32_t`.

Example code:

```
Serial.printf("RSSI: %d dBm\n", WiFi.RSSI());
```

Example output:

```
RSSI: -68 dBm
```

Connect Different

ESP8266 SDK provides alternate methods to connect ESP station to an access point. Out of them esp8266 / Arduino core implements WPS and Smart Config as described in more details below.

WPS

The following `beginWPSConfig` function allows connecting to a network using Wi-Fi Protected Setup (WPS). Currently only push-button configuration (`WPS_TYPE_PBC` mode) is supported (SDK 1.5.4).

```
WiFi.beginWPSConfig()
```

Depending on connection result function returns either `true` or `false` (`boolean` type).

Example code:

```
#include <ESP8266WiFi.h>

void setup(void)
{
  Serial.begin(115200);
  Serial.println();

  Serial.printf("Wi-Fi mode set to WIFI_STA %s\n", WiFi.mode(WIFI_STA) ? "" : "Failed!"
);
  Serial.print("Begin WPS (press WPS button on your router) ... ");
  Serial.println(WiFi.beginWPSConfig() ? "Success" : "Failed");

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {}
```

Example output:

```
Wi-Fi mode set to WIFI_STA
Begin WPS (press WPS button on your router) ... Success
.........
Connected, IP address: 192.168.1.102
```

Smart Config

The Smart Config connection of an ESP module an access point is done by sniffing for special packets that contain SSID and password of desired AP. To do so the mobile device or computer should have functionality of broadcasting of encoded SSID and password.

The following three functions are provided to implement Smart Config.

Start smart configuration mode by sniffing for special packets that contain SSID and password of desired Access Point. Depending on result either `true` or `false is returned.

```
beginSmartConfig()
```

Query Smart Config status, to decide when stop configuration. Function returns either `true` or `false of` boolean` type.

```
smartConfigDone()
```

Stop smart config, free the buffer taken by `beginSmartConfig()`. Depending on result function return either `true` or `false` of `boolean` type.

```
stopSmartConfig()
```

For additional details regarding Smart Config please refer to ESP8266 API User Guide.

ESP8266WiFi Library :back:

Station

Example of connecting to an access point has been shown in chapter Quick Start. In case connection is lost, ESP8266 will automatically reconnect to the last used access point, once it is again available.

Can we provide more robust connection to Wi-Fi than that?

Table of Contents

- Introduction
- Prepare Access Points
- Try it Out
- Can we Make it Simpler?
- Conclusion

Introduction

Following the example inQuick Start, we would like to go one step further and made ESP connect to next available access point if current connection is lost. This functionality is provided with 'ESP8266WiFiMulti' class and demonstrated in sketch below.

```
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>

ESP8266WiFiMulti wifiMulti;
boolean connectioWasAlive = true;

void setup()
{
  Serial.begin(115200);
  Serial.println();

  wifiMulti.addAP("primary-network-name", "pass-to-primary-network");
  wifiMulti.addAP("secondary-network-name", "pass-to-secondary-network");
  wifiMulti.addAP("tertiary-network-name", "pass-to-tertiary-network");
}

void monitorWiFi()
{
  if (wifiMulti.run() != WL_CONNECTED)
  {
    if (connectioWasAlive == true)
    {
      connectioWasAlive = false;
      Serial.print("Looking for WiFi ");
    }
    Serial.print(".");
    delay(500);
  }
  else if (connectioWasAlive == false)
  {
    connectioWasAlive = true;
    Serial.printf(" connected to %s\n", WiFi.SSID().c_str());
  }
}

void loop()
{
  monitorWiFi();
}
```

Prepare Access Points

To try this sketch in action you need two (or more) access points. In lines below replace `primary-network-name` and `pass-to-primary-network` with name and password to your primary network. Do the same for secondary network.

```
wifiMulti.addAP("primary-network-name", "pass-to-primary-network");
wifiMulti.addAP("secondary-network-name", "pass-to-secondary-network");
```

You may add more networks if you have more access points.

```
wifiMulti.addAP("tertiary-network-name", "pass-to-tertiary-network");
...
```

Try it Out

Now upload updated sketch to ESP module and open serial monitor. Module will first scan for available networks. Then it will select and connect to the network with stronger signal. In case connection is lost, module will connect to next one available.

This process may look something like:

```
Looking for WiFi ..... connected to sensor-net-1
Looking for WiFi ....... connected to sensor-net-2
Looking for WiFi .... connected to sensor-net-1
```

In above example ESP connected first to `sensor-net-1`. Then I have switched `sensor-net-1` off. ESP discovered that connection is lost and started searching for another configured network. That happened to be `sensor-net-2` so ESP connected to it. Then I have switched `sensor-net-1` back on and shut down `sensor-net-2`. ESP reconnected automatically to `sensor-net-1`.

Function `monitorWiFi()` is in place to show when connection is lost by displaying `Looking for WiFi`. Dots `....` are displayed during process of searching for another configured access point. Then a message like `connected to sensor-net-2` is shown when connection is established.

Can we Make it Simpler?

Please note that you may simplify this sketch by removing function `monitorWiFi()` and putting inside `loop()` only `wifiMulti.run()`. ESP will still reconnect between configured access points if required. Now you won't be able to see it on serial monitor unless you add `Serial.setDebugOutput(true)` as described in point Enable Wi-Fi Diagnostic.

Updated sketch for such scenario will look as follows:

```
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>

ESP8266WiFiMulti wifiMulti;

void setup()
{
  Serial.begin(115200);
  Serial.setDebugOutput(true);
  Serial.println();

  wifiMulti.addAP("primary-network-name", "pass-to-primary-network");
  wifiMulti.addAP("secondary-network-name", "pass-to-secondary-network");
  wifiMulti.addAP("tertiary-network-name", "pass-to-tertiary-network");
}

void loop()
{
  wifiMulti.run();
}
```

That's it! This is really all the code you need to make ESP automatically reconnecting between available networks.

After uploading sketch and opening the serial monitor, the messages will look as below.

Initial connection to sensor-net-1 on power up:

```
f r0, scandone
f r0, scandone
state: 0 -> 2 (b0)
state: 2 -> 3 (0)
state: 3 -> 5 (10)

add 0
aid 1
cnt
chg_B1:-40

connected with sensor-net-1, channel 1
dhcp client start...
ip:192.168.1.10,mask:255.255.255.0,gw:192.168.1.9
```

Lost connection to sensor-net-1 and establishing connection to sensor-net-2:

```
bcn_timout,ap_probe_send_start
ap_probe_send over, rest wifi status to disassoc
state: 5 -> 0 (1)
rm 0
f r-40, scandone
f r-40, scandone
f r-40, scandone
state: 0 -> 2 (b0)
state: 2 -> 3 (0)
state: 3 -> 5 (10)
add 0

aid 1
cnt

connected with sensor-net-2, channel 11
dhcp client start...
ip:192.168.1.102,mask:255.255.255.0,gw:192.168.1.234
```

Lost connection to sensor-net-2 and establishing connection back to sensor-net-1:

```
bcn_timout,ap_probe_send_start
ap_probe_send over, rest wifi status to disassoc
state: 5 -> 0 (1)
rm 0
f r-40, scandone
f r-40, scandone
f r-40, scandone
state: 0 -> 2 (b0)
state: 2 -> 3 (0)
state: 3 -> 5 (10)
add 0
aid 1
cnt

connected with sensor-net-1, channel 6
dhcp client start...
ip:192.168.1.10,mask:255.255.255.0,gw:192.168.1.9
```

Conclusion

I believe the minimalist sketch with `ESP8266WiFiMulti` class is a cool example what ESP8266 can do for us behind the scenes with just couple lines of code.

As shown in above example, reconnecting between access points takes time and is not seamless. Therefore, in practical applications, you will likely need to monitor connection status to decide e.g. if you can send the data to external system or should wait until connection is back.

For detailed review of functions provided to manage station mode please refer to the Station Class :arrow_right: documentation.

ESP8266WiFi Library :back:

Soft Access Point Class

Section below is ESP8266 specific as Arduino WiFi library documentation does not cover soft access point. The API description is broken down into three short chapters. They cover how to setup soft-AP, manage connection, and obtain information on soft-AP interface configuration.

Table of Contents

- Set up Network
  - softAP
  - softAPConfig
- Manage Network
  - softAPdisconnect
  - softAPgetStationNum
- Network Configuration
  - softAPIP
  - softAPmacAddress

Set up Network

This section describes functions to set up and configure ESP8266 in the soft access point (soft-AP) mode.

softAP

Set up a soft access point to establish a Wi-Fi network.

The simplest version (an overload in C++ terms) of this function requires only one parameter and is used to set up an open Wi-Fi network.

```
WiFi.softAP(ssid)
```

To set up password protected network, or to configure additional network parameters, use the following overload:

```
WiFi.softAP(ssid, password, channel, hidden)
```

The first parameter of this function is required, remaining three are optional.

Meaning of all parameters is as follows:

- `ssid` - character string containing network SSID (max. 63 characters)
- `password` - optional character string with a password. For WPA2-PSK network it should be at least 8 character long. If not specified, the access point will be open for anybody to connect.
- `channel` - optional parameter to set Wi-Fi channel, from 1 to 13. Default channel = 1.
- `hidden` - optional parameter, if set to `true` will hide SSID

Function will return `true` or `false` depending on result of setting the soft-AP.

Notes:

- The network established by softAP will have default IP address of 192.168.4.1. This address may be changed using `softAPConfig` (see below).
- Even though ESP8266 can operate in soft-AP + station mode, it actually has only one hardware channel. Therefore in soft-AP + station mode, the soft-AP channel will default to the number used by station. For more information how this may affect operation of stations connected to ESP8266's soft-AP, please check this FAQ entry on Espressif forum.

softAPConfig

Configure the soft access point's network interface.

```
softAPConfig (local_ip, gateway, subnet)
```

All parameters are the type of `IPAddress` and defined as follows:

- `local_ip` - IP address of the soft access point
- `gateway` - gateway IP address
- `subnet` - subnet mask

Function will return `true` or `false` depending on result of changing the configuration.

Example code:

```
#include <ESP8266WiFi.h>

IPAddress local_IP(192,168,4,22);
IPAddress gateway(192,168,4,9);
IPAddress subnet(255,255,255,0);

void setup()
{
  Serial.begin(115200);
  Serial.println();

  Serial.print("Setting soft-AP configuration ... ");
  Serial.println(WiFi.softAPConfig(local_IP, gateway, subnet) ? "Ready" : "Failed!");

  Serial.print("Setting soft-AP ... ");
  Serial.println(WiFi.softAP("ESPsoftAP_01") ? "Ready" : "Failed!");

  Serial.print("Soft-AP IP address = ");
  Serial.println(WiFi.softAPIP());
}

void loop() {}
```

Example output:

```
Setting soft-AP configuration ... Ready
Setting soft-AP ... Ready
Soft-AP IP address = 192.168.4.22
```

Manage Network

Once soft-AP is established you may check the number of stations connected, or shut it down, using the following functions.

softAPgetStationNum

Get the count of the stations that are connected to the soft-AP interface.

```
WiFi.softAPgetStationNum()
```

Example code:

```
Serial.printf("Stations connected to soft-AP = %d\n", WiFi.softAPgetStationNum());
```

Example output:

```
Stations connected to soft-AP = 2
```

Note: the maximum number of stations that may be connected to ESP8266 soft-AP is five.

softAPdisconnect

Disconnect stations from the network established by the soft-AP.

```
WiFi.softAPdisconnect(wifioff)
```

Function will set currently configured SSID and password of the soft-AP to null values. The parameter `wifioff` is optional. If set to `true` it will switch the soft-AP mode off.

Function will return `true` if operation was successful or `false` if otherwise.

Network Configuration

Functions below provide IP and MAC address of ESP8266's soft-AP.

softAPIP

Return IP address of the soft access point's network interface.

```
WiFi.softAPIP()
```

Returned value is of `IPAddress` type.

Example code:

```
Serial.print("Soft-AP IP address = ");
Serial.println(WiFi.softAPIP());
```

Example output:

```
Soft-AP IP address = 192.168.4.1
```

softAPmacAddress

Return MAC address of soft access point. This function comes in two versions, which differ in type of returned values. First returns a pointer, the second a `String`.

Pointer to MAC

```
WiFi.softAPmacAddress(mac)
```

Function accepts one parameter `mac` that is a pointer to memory location (an `uint8_t` array the size of 6 elements) to save the mac address. The same pointer value is returned by the function itself.

Example code:

```
uint8_t macAddr[6];
WiFi.softAPmacAddress(macAddr);
Serial.printf("MAC address = %02x:%02x:%02x:%02x:%02x:%02x\n", macAddr[0], macAddr[1],
 macAddr[2], macAddr[3], macAddr[4], macAddr[5]);
```

Example output:

```
MAC address = 5e:cf:7f:8b:10:13
```

MAC as a String

Optionally you can use function without any parameters that returns a `String` type value.

```
WiFi.softAPmacAddress()
```

Example code:

```
Serial.printf("MAC address = %s\n", WiFi.softAPmacAddress().c_str());
```

Example output:

```
MAC address = 5E:CF:7F:8B:10:13
```

For code samples please refer to separate section with examples :arrow_right: dedicated specifically to the Soft Access Point Class.

ESP8266WiFi Library :back:

Soft Access Point

Example below presents how to configure ESP8266 to run in soft access point mode so Wi-Fi stations can connect to it. The Wi-Fi network established by the soft-AP will be identified with the SSID set during configuration. The network may be protected with a password. The network may be also open, if no password is set during configuration.

Table of Contents

- The Sketch
- How to Use It?
- How Does it Work?
- Can we Make it Simpler?
- Conclusion

The Sketch

Setting up soft-AP with ESP8266 can be done with just couple lines of code.

```
#include <ESP8266WiFi.h>

void setup()
{
  Serial.begin(115200);
  Serial.println();

  Serial.print("Setting soft-AP ... ");
  boolean result = WiFi.softAP("ESPsoftAP_01", "pass-to-soft-AP");
  if(result == true)
  {
    Serial.println("Ready");
  }
  else
  {
    Serial.println("Failed!");
  }
}

void loop()
{
  Serial.printf("Stations connected = %d\n", WiFi.softAPgetStationNum());
  delay(3000);
}
```

How to Use It?

In line `boolean result = WiFi.softAP("ESPsoftAP_01", "pass-to-soft-AP")` change `pass-to-soft-AP` to some meaningful password and upload sketch. Open serial monitor and you should see:

```
Setting soft-AP ... Ready
Stations connected = 0
Stations connected = 0
...
```

Then take your mobile phone or a PC, open the list of available access points, find `ESPsoftAP_01` and connect to it. This should be reflected on serial monitor as a new station connected:

```
Stations connected = 1
Stations connected = 1
...
```

If you have another Wi-Fi station available then connect it as well. Check serial monitor again where you should now see two stations reported.

How Does it Work?

Sketch is small so analysis shouldn't be difficult. In first line we are including `ESP8266WiFi` library:

```
#include <ESP8266WiFi.h>
```

Setting up of the access point `ESPsoftAP_01` is done by executing:

```
 boolean result = WiFi.softAP("ESPsoftAP_01", "pass-to-soft-AP");
```

If this operation is successful then `result` will be `true` or `false` if otherwise. Basing on that either `Ready` or `Failed!` will be printed out by the following `if - else` conditional statement.

Can we Make it Simpler?

Can we make this sketch even simpler? Yes, we can! We can do it by using alternate `if - else` statement as below:

```
WiFi.softAP("ESPsoftAP_01", "pass-to-soft-AP") ? "Ready" : "Failed!"
```

Such statement will return either `Ready` or `Failed!` depending on result of `WiFi.softAP(...)`. This way we can considerably shorten our sketch without any changes to functionality:

```
#include <ESP8266WiFi.h>

void setup()
{
  Serial.begin(115200);
  Serial.println();

  Serial.print("Setting soft-AP ... ");
  Serial.println(WiFi.softAP("ESPsoftAP_01", "pass-to-soft-AP") ? "Ready" : "Failed!")
;
}

void loop()
{
  Serial.printf("Stations connected = %d\n", WiFi.softAPgetStationNum());
  delay(3000);
}
```

I believe this is very neat piece of code. If `? :` conditional operator is new to you, I recommend to start using it and make your code shorter and more elegant.

Conclusion

ESP8266WiFi library makes it easy to turn ESP8266 into soft access point.

Once you try above sketch check out WiFiAccessPoint.ino as a next step. It demonstrates how to access ESP operating in soft-AP mode from a web browser.

For the list of functions to manage ESP module in soft-AP mode please refer to the Soft Access Point Class :arrow_right: documentation.

ESP8266WiFi Library :back:

Scan Class

This class is represented in Arduino WiFi library by scanNetworks() function. Developers of esp8266 / Arduino core extend this functionality by additional methods and properties.

Table of Contents

- Scan for Networks
  - scanNetworks
  - scanNetworksAsync
  - scanComplete
  - scanDelete
- Show Results
  - SSID
  - encryptionType
  - BSSID
  - BSSIDstr
  - channel
  - isHidden
  - getNetworkInfo

Documentation of this class is divided into two parts. First covers functions to scan for available networks. Second describes what information is collected during scanning process and how to access it.

Scan for Networks

Scanning for networks takes hundreds of milliseconds to complete. This may be done in a single run when we are triggering scan process, waiting for completion, and providing result - all by a single function. Another option is to split this into steps, each done by a separate function. This way we can execute other tasks while scanning is in progress. This is called asynchronous scanning. Both methods of scanning are documented below.

scanNetworks

Scan for available Wi-Fi networks in one run and return the number of networks that has been discovered.

```
WiFi.scanNetworks()
```

There is on overload of this function that accepts two optional parameters to provide extended functionality of asynchronous scanning as well as looking for hidden networks.

```
WiFi.scanNetworks(async, show_hidden)
```

Both function parameters are of `boolean` type. They provide the flowing functionality:

- `asysnc` - if set to `true` then scanning will start in background and function will exit without waiting for result. To check for result use separate function `scanComplete` that is described below.
- `show_hidden` - set it to `true` to include in scan result networks with hidden SSID.

scanComplete

Check for result of asynchronous scanning.

```
WiFi.scanComplete()
```

On scan completion function returns the number of discovered networks.

If scan is not done, then returned value is < 0 as follows:

- Scanning still in progress: -1
- Scanning has not been triggered: -2

scanDelete

Delete the last scan result from memory.

```
WiFi.scanDelete()
```

scanNetworksAsync

Start scanning for available Wi-Fi networks. On completion execute another function.

```
WiFi.scanNetworksAsync(onComplete, show_hidden)
```

Function parameters:

- `onComplete` - the event handler executed when the scan is done
- `show_hidden` - optional `boolean` parameter, set it to `true` to scan for hidden networks

Example code:

```
#include "ESP8266WiFi.h"

void prinScanResult(int networksFound)
{
  Serial.printf("%d network(s) found\n", networksFound);
  for (int i = 0; i < networksFound; i++)
  {
    Serial.printf("%d: %s, Ch:%d (%ddBm) %s\n", i + 1, WiFi.SSID(i).c_str(), WiFi.chan
nel(i), WiFi.RSSI(i), WiFi.encryptionType(i) == ENC_TYPE_NONE ? "open" : "");
  }
}


void setup()
{
  Serial.begin(115200);
  Serial.println();

  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);

  WiFi.scanNetworksAsync(prinScanResult);
}


void loop() {}
```

Example output:

```
5 network(s) found
1: Tech_D005107, Ch:6 (-72dBm)
2: HP-Print-A2-Photosmart 7520, Ch:6 (-79dBm)
3: ESP_0B09E3, Ch:9 (-89dBm) open
4: Hack-4-fun-net, Ch:9 (-91dBm)
5: UPC Wi-Free, Ch:11 (-79dBm)
```

Show Results

Functions below provide access to result of scanning. It does not matter if scanning has been done in synchronous or asynchronous mode, scan results are available using the same API.

Individual results are accessible by providing a `networkItem' that identifies the index (zero based) of discovered network.

SSID

Return the SSID of a network discovered during the scan.

```
WiFi.SSID(networkItem)
```

Returned SSID is of the `String` type. The `networkItem` is a zero based index of network discovered during scan.

encryptionType

Return the encryption type of a network discovered during the scan.

```
WiFi.encryptionType(networkItem)
```

Function returns a number that encodes encryption type as follows:

- 5 : `ENC_TYPE_WEP` - WEP
- 2 : `ENC_TYPE_TKIP` - WPA / PSK
- 4 : `ENC_TYPE_CCMP` - WPA2 / PSK
- 7 : `ENC_TYPE_NONE` - open network
- 8 : `ENC_TYPE_AUTO` - WPA / WPA2 / PSK

The `networkItem` is a zero based index of network discovered during scan.

RSSI

Return the RSSI (Received Signal Strength Indication) of a network discovered during the scan.

```
WiFi.RSSI(networkItem)
```

Returned RSSI is of the `int32_t` type. The `networkItem` is a zero based index of network discovered during scan.

BSSID

Return the BSSID#Basic*service_set_identification*.28BSSID.29) (Basic Service Set Identification) that is another name of MAC address of a network discovered during the scan.

```
WiFi.BSSID(networkItem)
```

Function returns a pointer to the memory location (an `uint8_t` array with the size of 6 elements) where the BSSID is saved.

If you do not like to pointers, then there is another version of this function that returns a `String`.

```
WiFi.BSSIDstr(networkItem)
```

The `networkItem` is a zero based index of network discovered during scan.

channel

Return the channel of a network discovered during the scan.

```
WiFi.channel(networkItem)
```

Returned channel is of the `int32_t` type. The `networkItem` is a zero based index of network discovered during scan.

isHidden

Return information if a network discovered during the scan is hidden or not.

```
WiFi.isHidden(networkItem)
```

Returned value if the `bolean` type, and `true` means that network is hidden. The `networkItem` is a zero based index of network discovered during scan.

getNetworkInfo

Return all the network information discussed in this chapter above in a single function call.

```
WiFi.getNetworkInfo(networkItem, &ssid, &encryptionType, &RSSI, *&BSSID, &channel, &is
Hidden)
```

The `networkItem` is a zero based index of network discovered during scan. All other input parameters are passed to function by reference. Therefore they will be updated with actual values retrieved for particular `networkItem`. The function itself returns `boolean` `true` or `false` to confirm if information retrieval was successful or not.

Example code:

```
int n = WiFi.scanNetworks(false, true);

String ssid;
uint8_t encryptionType;
int32_t RSSI;
uint8_t* BSSID;
int32_t channel;
bool isHidden;

for (int i = 0; i < n; i++)
{
  WiFi.getNetworkInfo(i, ssid, encryptionType, RSSI, BSSID, channel, isHidden);
  Serial.printf("%d: %s, Ch:%d (%ddBm) %s %s\n", i + 1, ssid.c_str(), channel, RSSI, e
ncryptionType == ENC_TYPE_NONE ? "open" : "", isHidden ? "hidden" : "");
}
```

Example output:

```
6 network(s) found
1: Tech_D005107, Ch:6 (-72dBm)
2: HP-Print-A2-Photosmart 7520, Ch:6 (-79dBm)
3: ESP_0B09E3, Ch:9 (-89dBm) open
4: Hack-4-fun-net, Ch:9 (-91dBm)
5: , Ch:11 (-77dBm)  hidden
6: UPC Wi-Free, Ch:11 (-79dBm)
```

For code samples please refer to separate section with examples :arrow_right: dedicated specifically to the Scan Class.

ESP8266WiFi Library :back:

Scan

To connect a mobile phone to a hot spot, you typically open Wi-Fi settings app, list available networks and then pick the hot spot you need. You can also list the networks with ESP8266 and here is how.

Table of Contents

- Simple Scan
  - Disconnect
  - Scan for Networks
  - Complete Example
  - Example in Action
- Async Scan
  - No delay()
  - Setup
  - When to Start
  - Check When Done
  - Complete Example
  - Example in Action
- Conclusion

Simple Scan

This example shows the bare minimum code we need to check for the list of available networks.

Disconnect

To start with, enable module in station mode and then disconnect.

```
WiFi.mode(WIFI_STA);
WiFi.disconnect();
```

Running `WiFi.disconnect()` is to shut down a connection to an access point that module may have automatically made using previously saved credentials.

Scan for Networks

After some delay to let the module disconnect, go to scanning for available networks:

```
int n = WiFi.scanNetworks();
```

Now just check if returned `n` if greater than 0 and list found networks:

```
for (int i = 0; i < n; i++)
{
  Serial.println(WiFi.SSID(i));
}
```

This is that simple.

Complete Example

The sketch should have obligatory `#include <ESP8266WiFi.h>` and looks as follows:

```
#include "ESP8266WiFi.h"

void setup()
{
  Serial.begin(115200);
  Serial.println();

  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);
}

void loop()
{
  Serial.print("Scan start ... ");
  int n = WiFi.scanNetworks();
  Serial.print(n);
  Serial.println(" network(s) found");
  for (int i = 0; i < n; i++)
  {
    Serial.println(WiFi.SSID(i));
  }
  Serial.println();

  delay(5000);
}
```

Example in Action

Upload this sketch to ESP module and open a serial monitor. If there are access points around (sure there are) you will see a similar list repeatedly printed out:

```
Scan start ... 5 network(s) found
Tech_D005107
HP-Print-A2-Photosmart 7520
ESP_0B09E3
Hack-4-fun-net
UPC Wi-Free
```

When looking for the text `scan start ...` displayed, you will notice that it takes noticeable time for the following text `n network(s) found` to show up. This is because execution of `WiFi.scanNetworks()` takes time and our program is waiting for it to complete before moving to the next line of code. What if at the same time we would like ESP to run time critical process (e.g. animation) that should not be disturbed?

It turns out that this is fairly easy to do by scanning networks in async mode.

Check it out in next example below that will also demonstrate printing out other parameters of available networks besides SSID.

Async Scan

What we like to do, is to trigger process of scanning for networks and then return to executing code inside the `loop()`. Once scanning is complete, at a convenient time, we will check the list of networks. The "time critical process" will be simulated by a blinking LED at 250ms period.

We would like the blinking pattern not be disturbed at any time.

No delay()

To implement such functionality we should refrain from using any `delay()` inside the `loop()`. Instead we will define period when to trigger particular action. Then inside `loop()` we will check `millis()` (internal clock that counts milliseconds) and fire the action if the period expires.

Please check how this is done in BlinkWithoutDelay.ino example sketch. Identical technique can be used to periodically trigger scanning for Wi-Fi networks.

Setup

First we should define scanning period and internal variable `lastScanMillis` that will hold time when the last scan has been made.

```
#define SCAN_PERIOD 5000
long lastScanMillis;
```

When to Start

Then inside the `loop()` we will check if `SCAN_PERIOD` expired, so it is time to fire next scan:

```
if (currentMillis - lastScanMillis > SCAN_PERIOD)
{
  WiFi.scanNetworks(true);
  Serial.print("\nScan start ... ");
  lastScanMillis = currentMillis;
}
```

Please note that `WiFi.scanNetworks(true)` has an extra parameter `true` that was not present in previous example above. This is an instruction to scan in asynchronous mode, i.e. trigger scanning process, do not wait for result (processing will be done in background) and move to the next line of code. We need to use asynchronous mode otherwise 250ms LED blinking pattern would be disturbed as scanning takes longer than 250ms.

Check When Done

Finally we should periodically check for scan completion to print out the result once ready. To do so, we will use function `WiFi.scanComplete()`, that upon completion returns the number of found networks. If scanning is still in progress it returns -1. If scanning has not been triggered yet, it would return -2.

```
int n = WiFi.scanComplete();
if(n >= 0)
{
  Serial.printf("%d network(s) found\n", n);
  for (int i = 0; i < n; i++)
  {
    Serial.printf("%d: %s, Ch:%d (%ddBm) %s\n", i+1, WiFi.SSID(i).c_str(), WiFi.channe
l(i), WiFi.RSSI(i), WiFi.encryptionType(i) == ENC_TYPE_NONE ? "open" : "");
  }
  WiFi.scanDelete();
}
```

Please note function `WiFi.scanDelete()` that is deleting scanning result from memory, so it is not printed out over and over again on each `loop()` run.

Complete Example

Complete sketch is below. The code inside `setup()` is the same as described in previous example except for an additional `pinMode()` to configure the output pin for LED.

```
#include "ESP8266WiFi.h"

#define BLINK_PERIOD 250
long lastBlinkMillis;
boolean ledState;

#define SCAN_PERIOD 5000
long lastScanMillis;


void setup()
 {
  Serial.begin(115200);
  Serial.println();

  pinMode(LED_BUILTIN, OUTPUT);

  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);
}

void loop()
{
  long currentMillis = millis();

  // blink LED
  if (currentMillis - lastBlinkMillis > BLINK_PERIOD)
  {
    digitalWrite(LED_BUILTIN, ledState);
    ledState = !ledState;
    lastBlinkMillis = currentMillis;
  }

  // trigger Wi-Fi network scan
  if (currentMillis - lastScanMillis > SCAN_PERIOD)
  {
    WiFi.scanNetworks(true);
    Serial.print("\nScan start ... ");
    lastScanMillis = currentMillis;
  }

  // print out Wi-Fi network scan result uppon completion
  int n = WiFi.scanComplete();
  if(n >= 0)
  {
    Serial.printf("%d network(s) found\n", n);
```

```
    for (int i = 0; i < n; i++)
    {
      Serial.printf("%d: %s, Ch:%d (%ddBm) %s\n", i+1, WiFi.SSID(i).c_str(), WiFi.chan
nel(i), WiFi.RSSI(i), WiFi.encryptionType(i) == ENC_TYPE_NONE ? "open" : "");
    }
    WiFi.scanDelete();
  }
}
```

Example in Action

Upload above sketch to ESP module and open a serial monitor. You should see similar list printed out every 5 seconds:

```
Scan start ... 5 network(s) found
1: Tech_D005107, Ch:6 (-72dBm)
2: HP-Print-A2-Photosmart 7520, Ch:6 (-79dBm)
3: ESP_0B09E3, Ch:9 (-89dBm) open
4: Hack-4-fun-net, Ch:9 (-91dBm)
5: UPC Wi-Free, Ch:11 (-79dBm)
```

Check the LED. It should be blinking undisturbed four times per second.

Conclusion

The scan class API provides comprehensive set of methods to do scanning in both synchronous as well as in asynchronous mode. Therefore we can easy implement code that is doing scanning in background without disturbing other processes running on ESP8266 module.

For the list of functions provided to manage scan mode please refer to the Scan Class :arrow_right: documentation.

ESP8266WiFi Library :back:

Client Class

Methods documented for Client in Arduino

1. WiFiClient()
2. connected()
3. connect()
4. write()
5. print()
6. println()
7. available()
8. read()
9. flush()
10. stop()

Methods and properties described further down are specific to ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

setNoDelay

```
setNoDelay(nodelay)
```

With `nodelay` set to `true`, this function will to disable Nagle algorithm.

This algorithm is intended to reduce TCP/IP traffic of small packets sent over the network by combining a number of small outgoing messages, and sending them all at once. The downside of such approach is effectively delaying individual messages until a big enough packet is assembled.

Example:

```
clinet.setNoDelay(true);
```

Other Function Calls

```
uint8_t  status ()
virtual size_t  write (const uint8_t *buf, size_t size)
size_t  write_P (PGM_P buf, size_t size)
size_t  write (Stream &stream)
size_t  write (Stream &stream, size_t unitSize) __attribute__((deprecated))
virtual int  read (uint8_t *buf, size_t size)
virtual int  peek ()
virtual size_t  peekBytes (uint8_t *buffer, size_t length)
size_t  peekBytes (char *buffer, size_t length)
virtual  operator bool ()
IPAddress  remoteIP ()
uint16_t  remotePort ()
IPAddress  localIP ()
uint16_t  localPort ()
bool  getNoDelay ()
```

Documentation for the above functions is not yet prepared.

For code samples please refer to separate section with examples :arrow_right: dedicated specifically to the Client Class.

ESP8266WiFi Library :back:

Client

Let's write a simple client program to access a single web page and display its contents on a serial monitor. This is typical operation performed by a client to access server's API to retrieve specific information. For instance we may want to contact GitHub's API to periodically check the number of open issues reported on esp8266 / Arduino repository.

Table of Contents

- Introduction
- Get Connected to Wi-Fi
- Select a Server
- Instantiate the Client
- Get Connected to the Server
- Request the Data
- Read Reply from the Server
- Now to the Sketch
- Test it Live
- Test it More
- Conclusion

Introduction

This time we are going to concentrate just on retrieving a web page contents sent by a server, to demonstrate basic client's functionality. Once you are able to retrieve information from a server, you should be able to phrase it and extract specific data you need.

Get Connected to Wi-Fi

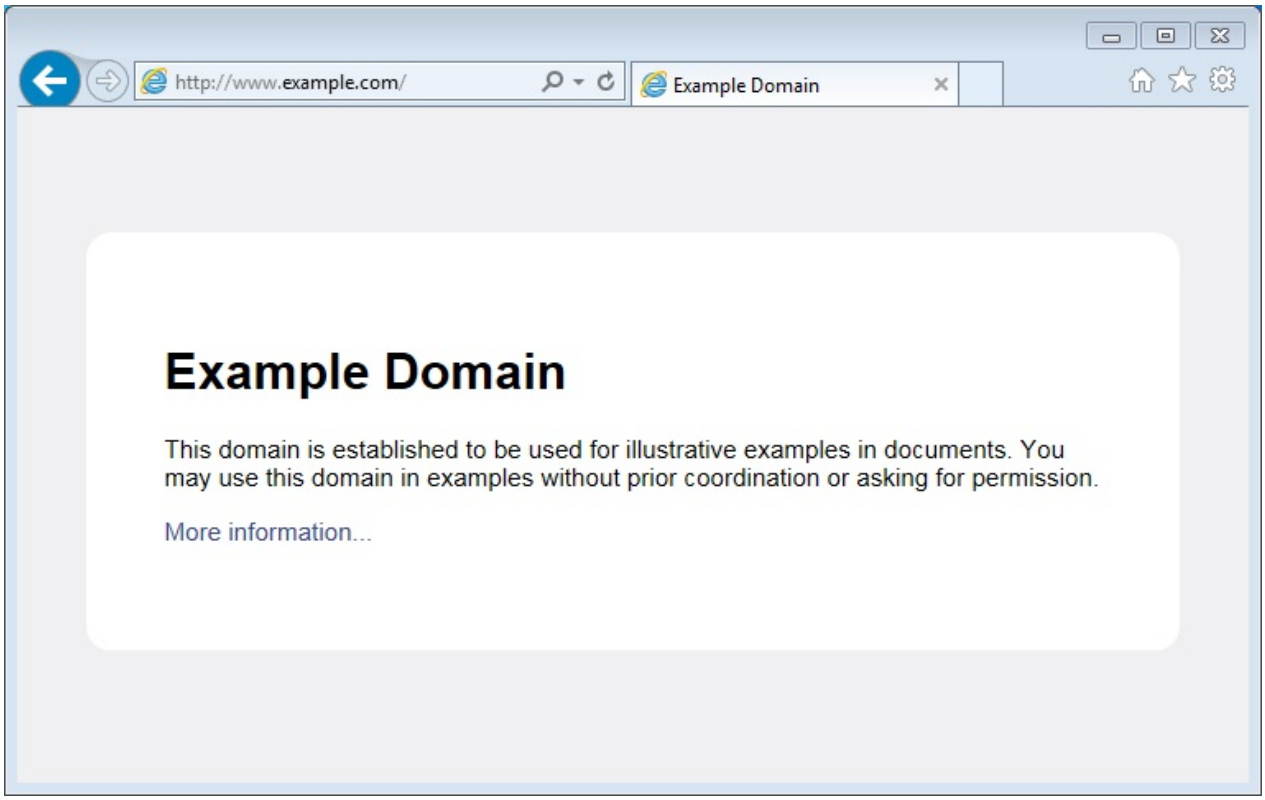
We should start with connecting the module to an access point to obtain an access to internet. The code to provide this functionality has been already discussed in chapter Quick Start. Please refer to it for details.

Select a Server

Once connected to the network we should connect to the specific server. Web address of this server is declared in `host` character string as below.

```
const char* host = "www.example.com";
```

I have selected `www.example.com` domain name and you can select any other. Just check if you can access it using a web browser.



Instantiate the Client

Now we should declare a client that will be contacting the host (server):

```
WiFiClient client;
```

Get Connected to the Server

In next line we will connect to the host and check the connection result. Note `80`, that is the standard port number used for web access.

```
if (client.connect(host, 80))
{
  // we are connected to the host!
}
else
{
  // connection failure
}
```

Request the Data

If connection is successful, we should send request the host to provide specific information we need. This is done using the HTTP GET request as in the following lines:

```
client.print(String("GET /") + " HTTP/1.1\r\n" +
             "Host: " + host + "\r\n" +
             "Connection: close\r\n" +
             "\r\n"
            );
```

Read Reply from the Server

Then, while connection by our client is still alive (`while (client.connected())` , see below) we can read line by line and print out server's response:

```
while (client.connected())
{
  if (client.available())
  {
    String line = client.readStringUntil('\n');
    Serial.println(line);
  }
}
```

The inner `if (client.available())` is checking if there are any data available from the server. If so, then they are printed out.

Once server sends all requested data it will disconnect and program will exit the `while` loop.

Now to the Sketch

Complete sketch, including a case when contention to the server fails, is presented below.

```
#include <ESP8266WiFi.h>

const char* ssid = "********";
const char* password = "********";

const char* host = "www.example.com";


void setup()
{
  Serial.begin(115200);
```

```
  Serial.println();

  Serial.printf("Connecting to %s ", ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println(" connected");
}


void loop()
{
  WiFiClient client;

  Serial.printf("\n[Connecting to %s ... ", host);
  if (client.connect(host, 80))
  {
    Serial.println("connected]");

    Serial.println("[Sending a request]");
    client.print(String("GET /") + " HTTP/1.1\r\n" +
                 "Host: " + host + "\r\n" +
                 "Connection: close\r\n" +
                 "\r\n"
                );

    Serial.println("[Response:]");
    while (client.connected())
    {
      if (client.available())
      {
        String line = client.readStringUntil('\n');
        Serial.println(line);
      }
    }
    client.stop();
    Serial.println("\n[Disconnected]");
  }
  else
  {
    Serial.println("connection failed!]");
    client.stop();
  }
  delay(5000);
}
```

Test it Live

Upload sketch the module and open serial monitor. You should see a log similar to presented below.

First, after establishing Wi-Fi connection, you should see confirmation, that client connected to the server and send the request:

```
Connecting to sensor-net ........ connected

[Connecting to www.example.com ... connected]
[Sending a request]
```

Then, after getting the request, server will first respond with a header that specifies what type of information will follow (e.g. `Content-Type: text/html`), how long it is (like `Content-Length: 1270`), etc.:

```
[Response:]
HTTP/1.1 200 OK

Cache-Control: max-age=604800
Content-Type: text/html
Date: Sat, 30 Jul 2016 12:30:45 GMT
Etag: "359670651+ident"
Expires: Sat, 06 Aug 2016 12:30:45 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (ewr/15BD)
Vary: Accept-Encoding
X-Cache: HIT
x-ec-custom-error: 1
Content-Length: 1270
Connection: close
```

End of header is marked with an empty line and then you should see the HTML code of requested web page.

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">

(...)

</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is established to be used for illustrative examples in documents. Y
ou may use this
    domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>

[Disconnected]
```

Test it More

In case server's web address is incorrect, or server is not accessible, you should see the following short and simple message on the serial monitor:

```
Connecting to sensor-net ........ connected

[Connecting to www.wrong-example.com ... connection failed!]
```

Conclusion

With this simple example we have demonstrated how to set up a client program, connect it to a server, request a web page and retrieve it. Now you should be able to write your own client program for ESP8266 and move to more advanced dialogue with a server, like e.g. using HTTPS protocol with the Client Secure.

For more client examples please check

- WiFiClientBasic.ino - a simple sketch that sends a message to a TCP server
- WiFiClient.ino - this sketch sends data via HTTP GET requests to data.sparkfun.com

service.

For the list of functions provided to manage clients, please refer to the Client Class :arrow_right: documentation.

ESP8266WiFi Library :back:

Client Secure Class

Methods and properties described in this section are specific to ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

loadCertificate

Load client certificate from file system.

```
loadCertificate(file)
```

This function and the following `setCertificate` is to load the client side certificate i.e. the certificate which ESP will provide when it connects to the server. It is applicable if the server uses client side certificate authentication — most don't use that.

Example code:

Declarations

```
#include <FS.h>
#include <ESP8266WiFi.h>
#include <WiFiClientSecure.h>

const char* certyficateFile = "/client.cer";
```

setup() or loop()

```
if (!SPIFFS.begin())
{
  Serial.println("Failed to mount the file system");
  return;
}

Serial.printf("Opening %s", certyficateFile);
File crtFile = SPIFFS.open(certyficateFile, "r");
if (!crtFile)
{
  Serial.println(" Failed!");
}

WiFiClientSecure client;

Serial.print("Loading %s", certyficateFile);
if (!client.loadCertificate(crtFile))
{
  Serial.println(" Failed!");
}

// proceed with connecting of client to the host
```

setCertificate

Load client certificate from C array.

```
setCertificate (array, size)
```

For a practical example please check this interesting blog.

Other Function Calls

```
bool  verify (const char *fingerprint, const char *domain_name)
void  setPrivateKey (const uint8_t *pk, size_t size)
bool  loadCertificate (Stream &stream, size_t size)
bool  loadPrivateKey (Stream &stream, size_t size)
template<typename TFile >  bool  loadPrivateKey (TFile &file)
```

Documentation for the above functions is not yet prepared.

For code samples please refer to separate section with examples :arrow_right: dedicated specifically to the Client Secure Class.

ESP8266WiFi Library :back:

Client Secure

The client secure is a client but secure. Application example below will be easier to follow if you check similar and simpler example for the "ordinary" client. That being said we will concentrate on discussing the code that is specific to the client secure.

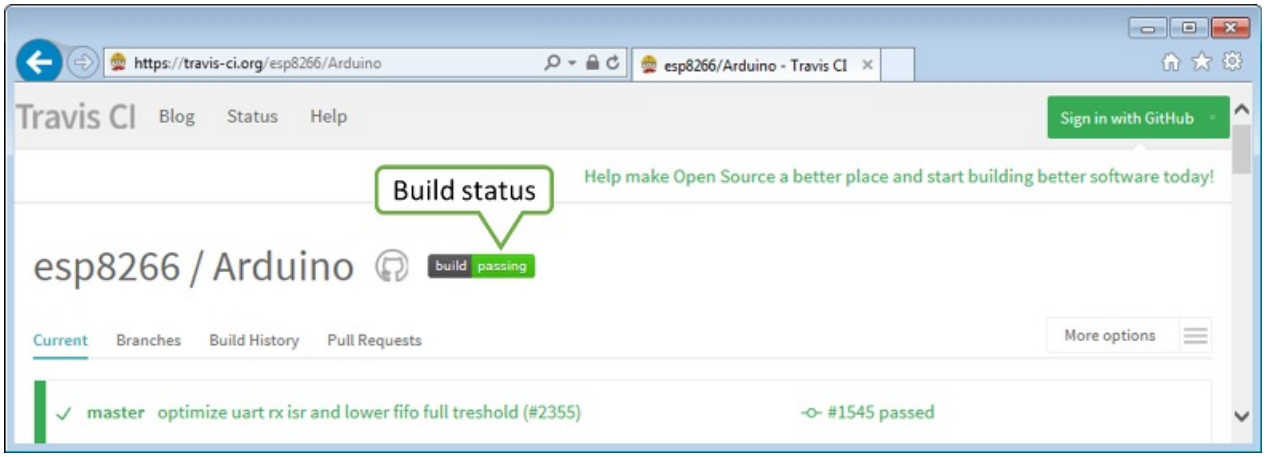
Table of Contents

- Introduction
- The Sketch
- How to Verify Server's Identity?
- Get the Fingerprint
- Connect to the Server
- Is it THAT Server?
- GET Response from the Server
- Read and Check the Response
- Does it Work?
- Conclusion

Introduction

In this example we will be retrieving information from a secure server https://api.github.com. This server is set up in place to provide specific and structured information on GitHub repositories. For instance, we may ask it to provide us the build status or the latest version of esp8266 / Adruino core.

The build status of esp8266 / Adruino may be checked on the repository's home page or on Travis CI site as below:



GitHub provides a separate server with API to access such information in structured form as JSON.

As you may guess we will use the client secure to contact https://api.github.com server and request the build status. If we open specific resource provided in the API with a web browser, the following should show up:

What we need to do, is to use client secure to connect to `https://api.github.com`, to GET `/repos/esp8266/Arduino/commits/master/status`, search for the line `"state": "success"` and display "Build Successful" if we find it, or "Build Failed" if otherwise.

The Sketch

A classic sketch HTTPSRequest.ino that is doing what we need is already available among examples of ESP8266WiFi library. Please open it to go through it step by step.

How to Verify Server's Identity?

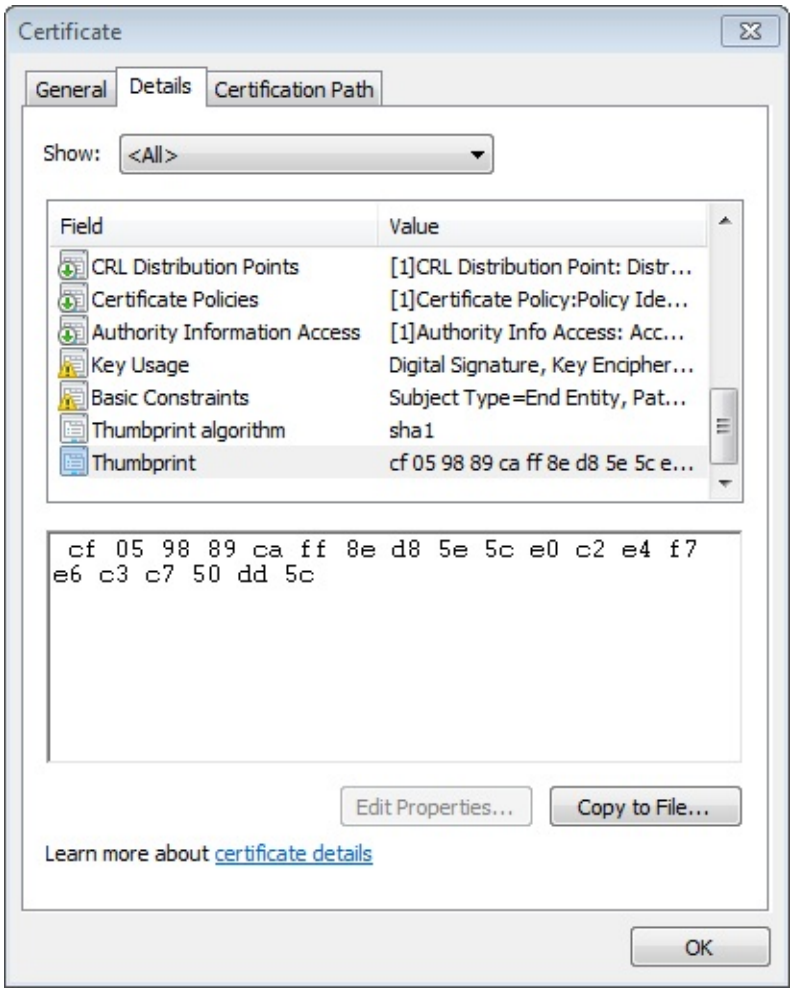
To establish a secure connection with a server we need to verify server's identity. Clients that run on "regular" computers do it by comparing server's certificate with locally stored list of trusted root certificates. Such certificates take several hundreds of KB, so it is not a good option for an ESP module. As an alternative we can use much smaller SHA1 fingerprint of specific certificate.

In declaration section of code we provide the name of `host` and the corresponding `fingerprint`.

```
const char* host = "api.github.com";
const char* fingerprint = "CF 05 98 89 CA FF 8E D8 5E 5C E0 C2 E4 F7 E6 C3 C7 50 DD 5C"
;
```

Get the Fingerprint

We can obtain the `fingerprint` for specific `host` using a web browser. For instance on Chrome press *Ctrl+Shift+I* and go to *Security* > *View Certificate* > *Details* > *Thumbprint*. This will show a window like below where you can copy the fingerprint and paste it into sketch.



Connect to the Server

Instantiate the `WiFiClientSecure` object and establish a connection (please note we need to use specific `httpsPort` for secure connections):

```
WiFiClientSecure client;
Serial.print("connecting to ");
Serial.println(host);
if (!client.connect(host, httpsPort)) {
  Serial.println("connection failed");
  return;
}
```

Is it THAT Server?

Now verify if the fingerprint we have matches this one provided by the server:

```
if (client.verify(fingerprint, host)) {
  Serial.println("certificate matches");
} else {
  Serial.println("certificate doesn't match");
}
```

If this check fails, it is up to you to decide if to proceed further or abort connection. Also note that certificates have specific validity period. Therefore the fingerprint of certificate we have checked today, will certainly be invalid some time later.

Remaining steps are identical as described in the non-secure client example.

GET Response from the Server

In the next steps we should execute GET command. This is done is similar way as discussed in non-secure client example.

```
client.print(String("GET ") + url + " HTTP/1.1\r\n" +
             "Host: " + host + "\r\n" +
             "User-Agent: BuildFailureDetectorESP8266\r\n" +
             "Connection: close\r\n\r\n");
```

After sending the request we should wait for a reply and then process received information.

Out of received replay we can skip response header. This can be done by reading until an empty line `"\r"` that marks the end of the header:

```
while (client.connected()) {
  String line = client.readStringUntil('\n');
  if (line == "\r") {
    Serial.println("headers received");
    break;
  }
}
```

Read and Check the Response

Finally we should read JSON provided by server and check if it contains `{"state": "success"` :

```
String line = client.readStringUntil('\n');
if (line.startsWith("{\"state\":\"success\"")) {
  Serial.println("esp8266/Arduino CI successfull!");
} else {
  Serial.println("esp8266/Arduino CI has failed");
}
```

Does it Work?

Now once you know how it should work, get the sketch. Update credentials to your Wi-Fi network. Check the current fingerprint of `api.github.com` and update it if required. Then upload sketch and open a serial monitor.

If everything is fine (including build status of esp8266 / Arduino) you should see message as below:

```
connecting to sensor-net
........
WiFi connected
IP address:
192.168.1.104
connecting to api.github.com
certificate matches
requesting URL: /repos/esp8266/Arduino/commits/master/status
request sent
headers received
esp8266/Arduino CI successfull!
reply was:
==========
{"state":"success","statuses":[{"url":"https://api.github.com/repos/esp8266/Arduino/st
atuses/8cd331a8bae04a6f1443ff0c93539af4720d8ddf","id":677326372,"state":"success","des
cription":"The Travis CI build passed","target_url":"https://travis-ci.org/esp8266/Ard
uino/builds/148827821","context":"continuous-integration/travis-ci/push","created_at":
"2016-08-01T09:54:38Z","updated_at":"2016-08-01T09:54:38Z"},{"url":"https://api.github
.com/repos/esp8266/Arduino/statuses/8cd331a8bae04a6f1443ff0c93539af4720d8ddf","id":677
333081,"state":"success","description":"27.62% (+0.00%) compared to 0718188","target_u
rl":"https://codecov.io/gh/esp8266/Arduino/commit/8cd331a8bae04a6f1443ff0c93539af4720d
8ddf","context":"codecov/project","created_at":"2016-08-01T09:59:05Z","updated_at":"20
16-08-01T09:59:05Z"},

(...)

==========
closing connection
```

Conclusion

Programming a secure client is almost identical as programming a non-secure client. The difference gets down to one extra step to verify server's identity. Keep in mind limitations due to heavy memory usage that depends on the strength of the key used by the server and whether server is willing to negotiate the TLS buffer size.

For the list of functions provided to manage secure clients, please refer to the Client Secure Class :arrow_right: documentation.

ESP8266WiFi Library :back:

Server Class

Methods documented for the Server Class in Arduino

1. WiFiServer()
2. begin()
3. available()
4. write()
5. print()
6. println()

Methods and properties described further down are specific to ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

setNoDelay

```
setNoDelay(nodelay)
```

With `nodelay` set to `true`, this function will to disable Nagle algorithm.

This algorithm is intended to reduce TCP/IP traffic of small packets sent over the network by combining a number of small outgoing messages, and sending them all at once. The downside of such approach is effectively delaying individual messages until a big enough packet is assembled.

Example:

```
server.begin();
server.setNoDelay(true);
```

Other Function Calls

```
bool  hasClient ()
bool  getNoDelay ()
virtual size_t  write (const uint8_t *buf, size_t size)
uint8_t  status ()
void  close ()
void  stop ()
```

Documentation for the above functions is not yet prepared.

For code samples please refer to separate section with examples :arrow_right: dedicated specifically to the Server Class.

ESP8266WiFi Library :back:

Server

Setting up web a server on ESP8266 requires very title code and is surprisingly straightforward. This is thanks to functionality provided by versatile ESP8266WiFi library.

The purpose of this example will be to prepare a web page that can be opened in a web browser. This page should show current raw reading of ESP's analog input pin.

Table of Contents

- The Object
- The Page
- Header First
- The Page is Served
- Get it Together
- Get it Run
- What Else?
- Conclusion

The Object

We will start off by creating a server object.

```
WiFiServer server(80);
```

The server responds to clients (in this case - web browsers) on port 80, which is a standard port web browsers talk to web servers.

The Page

Then let's write a short function `prepareHtmlPage()`, that will return a `String` class variable containing the contents of the web page. We will then pass this variable to server to pass it over to a client.

```
String prepareHtmlPage()
{
  String htmlPage =
     String("HTTP/1.1 200 OK\r\n") +
            "Content-Type: text/html\r\n" +
            "Connection: close\r\n" +  // the connection will be closed after completi
on of the response
            "Refresh: 5\r\n" +  // refresh the page automatically every 5 sec
            "\r\n" +
            "<!DOCTYPE HTML>" +
            "<html>" +
            "Analog input:  " + String(analogRead(A0)) +
            "</html>" +
            "\r\n";
  return htmlPage;
}
```

The function does nothing fancy but just puts together a text header and HTML contents of the page.

Header First

The header is to inform client what type of contents is to follow and how it will be served:

```
Content-Type: text/html
Connection: close
Refresh: 5
```

In our example the content type is `text/html`, the connection will be closed after serving and the content should requested by client again every 5 seconds. The header is concluded with an empty line `\r\n`. This is to distinguish header from the content to follow.

```
<!DOCTYPE HTML>
<html>
Analog input:  [Value]
</html>
```

The content contains two basic HTML tags, one to denote HTML document type `<!DOCTYPE HTML>` and another to mark beginning `<html>` and end `</html>` of the document. Inside there is a raw value read from ESP's analog input `analogRead(A0)` converted to the `String` type.

```
String(analogRead(A0))
```

The Page is Served

Serving of this web page will be done in the `loop()` where server is waiting for a new client to connect and send some data containing a request:

```
void loop()
{
  WiFiClient client = server.available();
  if (client)
  {
    // we have a new client sending some request
  }
}
```

Once a new client is connected, server will read the client's request and print it out on a serial monitor.

```
while (client.connected())
{
  if (client.available())
  {
    String line = client.readStringUntil('\r');
    Serial.print(line);
  }
}
```

Request from the client is marked with an empty new line. If we find this mark, we can send back the web page and exit `while()` loop using `break`.

```
if (line.length() == 1 && line[0] == '\n')
{
    client.println(prepareHtmlPage());
    break;
}
```

The whole process is concluded by stopping the connection with client:

```
client.stop();
```

Put it Together

Complete sketch is presented below.

```
#include <ESP8266WiFi.h>
```

```
const char* ssid = "********";
const char* password = "********";

WiFiServer server(80);


void setup()
{
  Serial.begin(115200);
  Serial.println();

  Serial.printf("Connecting to %s ", ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println(" connected");

  server.begin();
  Serial.printf("Web server started, open %s in a web browser\n", WiFi.localIP().toStr
ing().c_str());
}


// prepare a web page to be send to a client (web browser)
String prepareHtmlPage()
{
  String htmlPage =
     String("HTTP/1.1 200 OK\r\n") +
            "Content-Type: text/html\r\n" +
            "Connection: close\r\n" +  // the connection will be closed after completi
on of the response
            "Refresh: 5\r\n" +  // refresh the page automatically every 5 sec
            "\r\n" +
            "<!DOCTYPE HTML>" +
            "<html>" +
            "Analog input:  " + String(analogRead(A0)) +
            "</html>" +
            "\r\n";
  return htmlPage;
}


void loop()
{
  WiFiClient client = server.available();
  // wait for a client (web browser) to connect
  if (client)
  {
    Serial.println("\n[Client connected]");
```

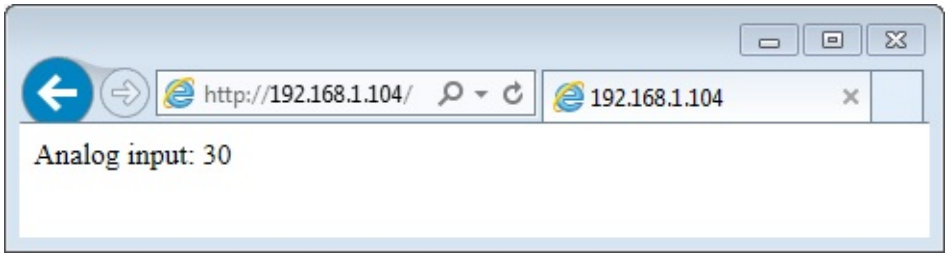
```
    while (client.connected())
    {
      // read line by line what the client (web browser) is requesting
      if (client.available())
      {
        String line = client.readStringUntil('\r');
        Serial.print(line);
        // look for the for end of client's request, that is marked with an empty line
        if (line.length() == 1 && line[0] == '\n')
        {
          client.println(prepareHtmlPage());
          break;
        }
      }
    }
    delay(1); // give the web browser time to receive the data

    // close the connection:
    client.stop();
    Serial.println("[Client disonnected]");
  }
}
```

Get it Run

Update `ssid` and `password` in sketch to match credentials of your access point. Load sketch to ESP module and open a serial monitor. First you should see confirmation that module connected to the access point and the web server started.

```
Connecting to sensor-net ........ connected
Web server started, open 192.168.1.104 in a web browser
```

Enter provided IP address in a web browser. You should see the page served by ESP8266:



The page would be refreshed every 5 seconds. Each time this happens, you should see request from the client (your web browser) printed out on the serial monitor:

```
[Client connected]
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept-Encoding: gzip, deflate
Host: 192.168.1.104
DNT: 1
Connection: Keep-Alive
[client disonnected]
```

What Else?

Looking on client examples you will quickly find out the similarities in protocol to the server. The protocol starts with a header that contains information what communication will be about. It contains what content type is communicated or accepted like `text/html`. It states whether connection will be kept alive or closed after submission of the header. It contains identification of the sender like `User-Agent: Mozilla/5.0 (Windows NT 6.1)`, etc.

Conclusion

The above example shows that a web server on ESP8266 can be set up in almost no time. Such server can easily stand up requests from much more powerful hardware and software like a PC with a web browser. Check out other classes like ESP8266WebServer that let you program more advanced applications.

If you like to try another server example, check out WiFiWebServer.ino, that provides functionality of toggling the GPIO pin on and off out of a web browser.

For the list of functions provided to implement and manage servers, please refer to the Server Class :arrow_right: documentation.

ESP8266WiFi Library :back:

UDP Class

Methods documented for WiFiUDP Class in Arduino

1. begin()
2. available()
3. beginPacket()
4. endPacket()
5. write()
6. parsePacket()
7. peek()
8. read()
9. flush()
10. stop()
11. remoteIP()
12. remotePort()

Methods and properties described further down are specific to ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

Multicast UDP

```
uint8_t  beginMulticast (IPAddress interfaceAddr, IPAddress multicast, uint16_t port)
virtual int  beginPacketMulticast (IPAddress multicastAddress, uint16_t port, IPAddres
s interfaceAddress, int ttl=1)
IPAddress  destinationIP ()
uint16_t  localPort ()
```

The `WiFiUDP` class supports sending and receiving multicast packets on STA interface. When sending a multicast packet, replace `udp.beginPacket(addr, port)` with `udp.beginPacketMulticast(addr, port, WiFi.localIP())`. When listening to multicast packets, replace `udp.begin(port)` with `udp.beginMulticast(WiFi.localIP(), multicast_ip_addr, port)`. You can use `udp.destinationIP()` to tell whether the packet received was sent to the multicast or unicast address.

For code samples please refer to separate section with examples :arrow_right: dedicated specifically to the UDP Class.

ESP8266WiFi Library :back:

UDP

The purpose of example application below is to demonstrate UDP communication between ESP8266 and an external client. The application (performing the role of a server) is checking inside the `loop()` for an UDP packet to arrive. When a valid packet is received, an acknowledge packet is sent back to the client to the same port it has been sent out.

Table of Contents

- Declarations
- Wi-Fi Connection
- UDP Setup
- An UDP Packet Arrived!
- An Acknowledge Send Out
- Complete Sketch
- How to Check It?
- Conclusion

Declarations

At the beginning of sketch we need to include two libraries:

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
```

The first library `ESP8266WiFi.h` is required by default if we are using ESP8266's Wi-Fi. The second one `WiFiUdp.h` is needed specifically for programming of UDP routines.

Once we have libraries in place we need to create a `WiFiUDP` object. Then we should specify a port to listen to incoming packets. There are conventions on usage of port numbers, for information please refer to the List of TCP and UDP port numbers. Finally we need to set up a buffer for incoming packets and define a reply message.

```
WiFiUDP Udp;
unsigned int localUdpPort = 4210;
char incomingPacket[255];
char  replyPacekt[] = "Hi there! Got the message :-)";
```

Wi-Fi Connection

At the beginning of `setup()` let's implement typical code to connect to an access point. This has been discussed in Quick Start. Please refer to it if required.

UDP Setup

Once connection is established, you can start listening to incoming packets.

```
Udp.begin(localUdpPort);
```

That is all required preparation. We can move to the `loop()` that will be handling actual UDP communication.

An UDP Packet Arrived!

Waiting for incoming UDP packed is done by the following code:

```
int packetSize = Udp.parsePacket();
if (packetSize)
{
  Serial.printf("Received %d bytes from %s, port %d\n", packetSize, Udp.remoteIP().toS
tring().c_str(), Udp.remotePort());
  int len = Udp.read(incomingPacket, 255);
  if (len > 0)
  {
    incomingPacket[len] = 0;
  }
  Serial.printf("UDP packet contents: %s\n", incomingPacket);

  (...)
}
```

Once a packet is received, the code will printing out the IP address and port of the sender as well as the length of received packet. If the packet is not empty, its contents will be printed out as well.

An Acknowledge Send Out

For each received packet we are sending back an acknowledge packet:

```
Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
Udp.write(replyPacekt);
Udp.endPacket();
```

Please note we are sending reply to the IP and port of the sender by using `Udp.remoteIP()` and `Udp.remotePort()` .

Complete Sketch

The sketch performing all described functionality is presented below:

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

const char* ssid = "********";
const char* password = "********";

WiFiUDP Udp;
unsigned int localUdpPort = 4210;  // local port to listen on
char incomingPacket[255];  // buffer for incoming packets
char  replyPacekt[] = "Hi there! Got the message :-)";  // a reply string to send back


void setup()
{
  Serial.begin(115200);
  Serial.println();

  Serial.printf("Connecting to %s ", ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println(" connected");

  Udp.begin(localUdpPort);
  Serial.printf("Now listening at IP %s, UDP port %d\n", WiFi.localIP().toString().c_s
tr(), localUdpPort);
}


void loop()
{
  int packetSize = Udp.parsePacket();
  if (packetSize)
  {
    // receive incoming UDP packets
    Serial.printf("Received %d bytes from %s, port %d\n", packetSize, Udp.remoteIP().t
oString().c_str(), Udp.remotePort());
    int len = Udp.read(incomingPacket, 255);
    if (len > 0)
    {
      incomingPacket[len] = 0;
```

```
    }
    Serial.printf("UDP packet contents: %s\n", incomingPacket);

    // send back a reply, to the IP address and port we got the packet from
    Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
    Udp.write(replyPacekt);
    Udp.endPacket();
  }
}
```

How to Check It?

Upload sketch to module and open serial monitor. You should see confirmation that ESP has connected to Wi-Fi and started listening to UDP packets:

```
Connecting to twc-net-3 ........ connected
Now listening at IP 192.168.1.104, UDP port 4210
```

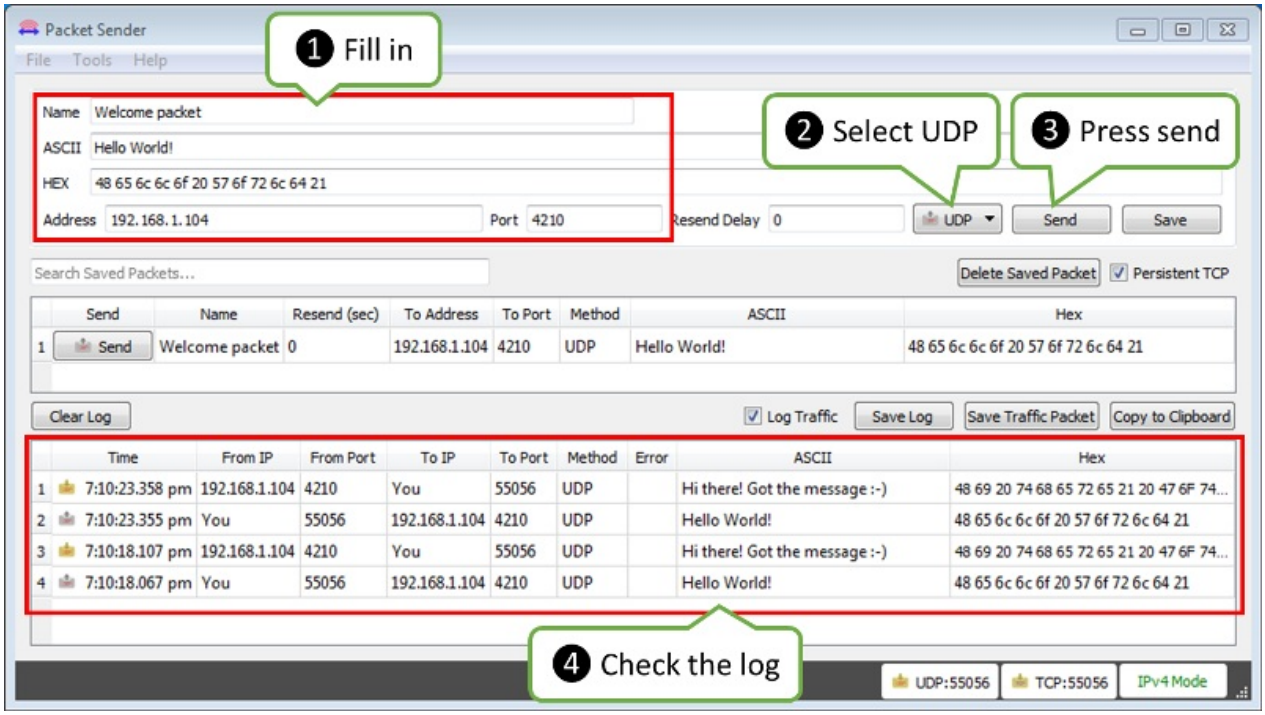
Now we need another application to send some packets to IP and port shown by ESP above.

Instead of programming another ESP, let's make it easier and use a purpose build application. I have selected the Packet Sender. It is available for popular operating systems. Download, install and execute it.

Once Packet Sender's window show up enter the following information:

- *Name* of the packet
- *ASCII* text of the message to be send inside the packet
- IP *Address* shown by our ESP
- *Port* shown by the ESP
- Select *UDP*

What I have entered is shown below:



Now click *Send*.

Immediately after that you should see the following on ESP's serial monitor:

```
Received 12 bytes from 192.168.1.106, port 55056
UDP packet contents: Hello World!
```

The text `192.168.1.106, port 55056` identifies a PC where the packet is send from. You will likely see different values.

As ESP sends an acknowledge packet back, you should see it in the log in the bottom part of the Packet Sender's window.

Conclusion

This simple example shows how to send and receive UDP packets between ESP and an external application. Once tested in this minimal set up, you should be able to program ESP to talk to any other UDP device. In case of issues to establish communication with a new device, use the Packet Sender or other similar program for troubleshooting

For review of functions provided to send and receive UDP packets, please refer to the UDP Class :arrow_right: documentation.

ESP8266WiFi Library :back:

Generic Class

Methods and properties described in this section are specific to ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

onEvent

```
void  onEvent (WiFiEventCb cb, WiFiEvent_t event=WIFI_EVENT_ANY) __attribute__((deprec
ated))
```

To see how to use `onEvent` please check example sketch WiFiClientEvents.ino available inside examples folder of the ESP8266WiFi library.

WiFiEventHandler

```
WiFiEventHandler  onStationModeConnected (std::function< void(const WiFiEventStationMo
deConnected &)>)
WiFiEventHandler  onStationModeDisconnected (std::function< void(const WiFiEventStatio
nModeDisconnected &)>)
WiFiEventHandler  onStationModeAuthModeChanged (std::function< void(const WiFiEventSta
tionModeAuthModeChanged &)>)
WiFiEventHandler  onStationModeGotIP (std::function< void(const WiFiEventStationModeGo
tIP &)>)
WiFiEventHandler  onStationModeDHCPTimeout (std::function< void(void)>)
WiFiEventHandler  onSoftAPModeStationConnected (std::function< void(const WiFiEventSof
tAPModeStationConnected &)>)
WiFiEventHandler  onSoftAPModeStationDisconnected (std::function< void(const WiFiEvent
SoftAPModeStationDisconnected &)>)
```

To see a sample application with `WiFiEventHandler`, please check separate section with examples :arrow_right: dedicated specifically to the Generic Class..

persistent

```
WiFi.persistent (persistent)
```

Module is able to reconnect to last used Wi-Fi network on power up or reset basing on settings stored in specific sectors of flash memory. By default these settings are written to flash each time they are used in functions like `WiFi.begin(ssid, password)`. This happens no matter if SSID or password has been actually changed.

This might result in some wear of flash memory depending on how often such functions are called.

Setting `persistent` to `false` will get SSID / password written to flash only if currently used values do not match what is already stored in flash.

Please note that functions `WiFi.disconnect` or `WiFi.softAPdisconnect` reset currently used SSID / password. If `persistent` is set to `false`, then using these functions will not affect SSID / password stored in flash.

To learn more about this functionality, and why it has been introduced, check issue report #1054.

mode

```
WiFi.mode(m)
WiFi.getMode()
```

- `WiFi.mode(m)`: set mode to `WIFI_AP`, `WIFI_STA`, `WIFI_AP_STA` or `WIFI_OFF`
- `WiFi.getMode()`: return current Wi-Fi mode (one out of four modes above)

Other Function Calls

```
int32_t  channel (void)
bool  setSleepMode (WiFiSleepType_t type)
WiFiSleepType_t  getSleepMode ()
bool  setPhyMode (WiFiPhyMode_t mode)
WiFiPhyMode_t  getPhyMode ()
void  setOutputPower (float dBm)
WiFiMode_t  getMode ()
bool  enableSTA (bool enable)
bool  enableAP (bool enable)
bool  forceSleepBegin (uint32 sleepUs=0)
bool  forceSleepWake ()
int  hostByName (const char *aHostname, IPAddress &aResult)
```

Documentation for the above functions is not yet prepared.

For code samples please refer to separate section with examples :arrow_right: dedicated specifically to the Generic Class.

ESP8266WiFi Library :back:

Generic

In the first example of the ESP8266WiFi library documentation we have discussed how to check when module connects to the Wi-Fi network. We were waiting until connection is established. If network is not available, the module could wait like that for ever doing nothing else. Another example on the Wi-Fi asynchronous scan mode demonstrated how to wait for scan result and do in parallel something else - blink a LED not disturbing the blink pattern. Let's apply similar functionality when connecting the module to an access point.

Table of Contents

- Introduction
- What are the Tasks?
- Event Driven Methods
- Register the Events
- The Code
- Check the Code
- Conclusion

Introduction

In example below we will show another cool example of getting ESP perform couple of tasks at the same time and with very little programming.

What are the Tasks?

We would like to write a code that will inform us that connection to Wi-Fi network has been established or lost. At the same time we want to perform some time critical task. We will simulate it with a blinking LED. Generic class provides specific, event driven methods, that will be executed asynchronously, depending on e.g. connection status, while we are already doing other tasks.

Event Driven Methods

The list of all such methods is provided in Generic Class documentation.

We would like to use two of them:

- `onStationModeGotIP` called when station is assigned IP address. This assignment may be done by DHCP client or by executing `WiFi.config(...)`.
- `onStationModeDisconnected` called when station is disconnected from Wi-Fi network. The reason of disconnection does not matter. Event will be triggered both if disconnection is done from the code by executing `WiFi.disconnect()`, because the Wi-Fi signal is weak, or because the access point is switched off.

Register the Events

To get events to work we need to complete just two steps:

1. Declare the event handler:

   ```
   WiFiEventHandler disconnectedEventHandler;
   ```

2. Select particular event (in this case `onStationModeDisconnected`) and add the code to be executed when event is fired.

   ```
   disconnectedEventHandler = WiFi.onStationModeDisconnected([](const WiFiEventStatio
   nModeDisconnected& event)
   {
    Serial.println("Station disconnected");
   });
   ```

   If this event is fired the code will print out information that station has been disconnected.

That's it. It is all we need to do.

The Code

The complete code, including both methods discussed at the beginning, is provided below.

```
#include <ESP8266WiFi.h>

const char* ssid = "********";
const char* password = "********";

WiFiEventHandler gotIpEventHandler, disconnectedEventHandler;

bool ledState;


void setup()
{
  Serial.begin(115200);
  Serial.println();

  pinMode(LED_BUILTIN, OUTPUT);

  gotIpEventHandler = WiFi.onStationModeGotIP([](const WiFiEventStationModeGotIP& even
t)
  {
    Serial.print("Station connected, IP: ");
    Serial.println(WiFi.localIP());
  });

  disconnectedEventHandler = WiFi.onStationModeDisconnected([](const WiFiEventStationM
odeDisconnected& event)
  {
    Serial.println("Station disconnected");
  });

  Serial.printf("Connecting to %s ...\n", ssid);
  WiFi.begin(ssid, password);
}


void loop()
{
  digitalWrite(LED_BUILTIN, ledState);
  ledState = !ledState;
  delay(250);
}
```

Check the Code

After uploading above sketch and opening a serial monitor we should see a similar log:

```
Connecting to sensor-net ...
Station connected, IP: 192.168.1.10
```

If you switch off the access point, and put it back on, you will see the following:

```
Station disconnected
Station disconnected
Station disconnected
Station connected, IP: 192.168.1.10
```

The process of connection, disconnection and printing messages is done in background of the `loop()` that is responsible for blinking the LED. Therefore the blink pattern all the time remains undisturbed.

Conclusion

Check out events from generic class. They will help you to write more compact code. Use them to practice splitting your code into separate tasks that are executed asynchronously.

For review of functions included in generic class, please refer to the Generic Class :arrow_right: documentation.